Alberto Ferazzoli

Babel Obfuscator

User's Guide

Babel Obfuscator User's Guide

Manual Version 3.5.0.0
http://www.babelfor.net/

Trademarks

Warning and Disclaimer

Every effort has been made to make this manual as complete and accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this manual.

Table of Contents

Introduction

Thank you for choosing Babel Obfuscator for the *.NET Framework*. This manual will guide you through the use of the numerous features of this obfuscator. *Babel* was made with the goal of being easy to use and at the same time deliver powerful obfuscation features. This is a *CLI* (Command Line Interface) driven tool, so there is no user interface. Since this tool integrates well with *Visual Studio* and *MSBuild,* the lack of a UI will not be a problem for most users.

Obfuscation at Glance

Obfuscation is a transformation process in which the code is changed to make it unclear and difficult to understand, so that reversing is more difficult. Software written in .NET languages like *C#* and *Visual Basic* are usually easy to reverse engineer because they compile to *MSIL* (Microsoft Intermediate Language), a CPU- independent instruction set that is embedded into .NET assemblies, along with other information (*Metadata*). This enables decompilation back to the original source code.

There are plenty of tools available online that enable the decompilation of .NET binaries into high-level languages. The most popular are *Reflector* and *IDA*. And the Microsoft .NET Framework SDK provides *ILDASM*, a tool to disassemble .NET binaries to MSIL, making reverse engineering extremely easy. Since MSIL is CPU independent, .NET assemblies can run on any platform that implements the CLR (Common Language Runtime)—a software infrastructure necessary to execute .NET binaries. The drawback is that software companies and IT professionals cannot protect their intellectual property if their software can be decompiled.

Benefits of Babel Obfuscator

Babel Obfuscator is a tool for the .NET Framework to transform assemblies in order to conceal the code and make reverse engineering difficult. The completely managed solution of MSIL Encryption ensures that the code of the method is no longer accessible using the disassemblers available on the market. MSIL encryption can be configured to strip code from the assembly and serialize it to encrypted files that can be loaded at runtime from safe stores or license files. This method permits the deployment of your software with a reduced set of features in the primary binaries in which the functionality can be activated only by access to the safe store.

Babel also performs user string encryption and control flow obfuscation, hiding all the information that can be used by reversers to identify and break software protection routines.

Symbol overload renaming reduces the resulting metadata size, which also reduces the overall size of the obfuscated assembly and improves executable load time.

The integration with *MSBuild* enables *Babel* to be used in automated build environments, and thus significantly reduces the time required to deliver the final product to testing and deployment.

Important note: A lot of effort was invested to ensure that the transformations performed by Babel are safe, and the resulting software functionalities are not affected. Since obfuscation processes involve complex MSIL changes, occasionally the obfuscated assembly could be compromised. To avoid this possibility the obfuscated software should be thoroughly tested before deploying to the end users.

Software Requirements

The following software is required:

- A version of Windows that supports the .NET Framework 3.5. This can be Windows XP SP2, Windows Server 2003 SP1 (including the R2 edition), Windows Vista and Windows 7.
- .NET Framework 3.5 SP1, which is installed by default starting with Windows Vista
- The Windows Software Development Kit (SDK), specifically the .NET tools it includes. This is a free download from http://msdn.com.

In addition, the following software is recommended:

- *Visual Studio 2008* or later; an alternative free *Express* edition can be downloaded from *http://msdn.com*

Conventions Used in This Manual

Various typefaces in this manual identify terms and other special items.
These typefaces include the following:

Typeface	Meaning
Italic	*Italic* is used for new terms or phrases when they are initially defined and occasionally for emphasis.
`Monospace`	`Monospace is used for screen messages, code listings, command samples, and filenames.` Code listings are colorized similarly to *Visual Studio.* `Blue monospace` type is used for XML elements and *C#* keywords. **`Brown monospace`** type is used for XML element names and *C#* strings. `Green monospace` type is used for comments. `Red monospace` type is used for XML attributes. `Teal monospace` type is used for C# type names.

Features

Babel Obfuscator is a powerful tool with many features available in three different editions. The most feature-rich edition is the *Enterprise* edition. This is the only version that enables Assembly Merging and MSIL encryption. There is also a *Professional* edition and a *Free* edition with a reduced set of features. To see all the supported features that come with each edition please refer to the Feature Matrix.

The main features are listed below:

- Works with Microsoft NET Framework 1.1, 2.0, 3.5, 4.0, Silverlight, Compact Framework
- Obfuscates Namespaces, Types, Methods, Events, Properties and Fields
- Unicode Normalization
- Includes Generic Types and Virtual Function Obfuscation
- Assembly Merging
- Assembly Embedding
- Automatic Obfuscation of Satellite Assemblies
- MSIL Control Flow Obfuscation
- String Encryption
- MSIL Encryption
- Embedded Resources Encryption
- Dead Code Removal
- Selective Obfuscation with XML Rule Files
- XML Mapping Files
- Declarative Obfuscation using Custom Attributes
- Public Symbol Obfuscation
- Silverlight XAP Package Obfuscation
- *Visual Studio* Post-Build Integration
- *MSBuild* Integration
- Supports Multiprocessor Execution
- Command Line Interface
- Supports re-sign with PFX and Strong Name Signature
- Disables tools like Reflector, Reflexil plug-in, and ILDASM

What's New

Babel 3.5.0.0 adds new important features:

- Support for .NET Framework 4.0
- Support for Compact Framework
- Embedded Resources Encryption
- Assembly Embedding
- Protection Against Disassemblers
- Added Hash String Algorithm for Silverlight

Details are available at

http://www.babelfor.net/ReleaseNotes.aspx

Babel Obfuscator was available for free until release 2.0.0.1. The 3.5.0.0 release introduces some major improvements and is available under a commercial license. The features introduced in this new version that are not available in Babel 2.0.0.1 are:

- Support for .NET Framework 4.0
- Extended Command Line Help
- 16 New Command Line Options
- Silverlight XAP Package Support
- XAML Parsing for Silverlight Applications
- Assembly Merging
- Assembly Embedding
- Embedded Resources Encryption
- MSIL Encryption
- Anti Reflection Tricks
- Automatic Obfuscation of Satellite Assemblies
- Public Symbol Obfuscation
- Revisited String Encryption
- Revisited Control Flow Obfuscation
- Improved Overload Renaming
- Extended XML Rules Files
- PFX File Support
- Multiprocessor Execution
- Performance Improvements
- Major Fixes on Symbol Renaming and Metadata Verification
- Log File Output

Feature Matrix

Babel is available in 3 editions: *Free*, *Professional* and *Enterprise*. The following table shows the features included in each edition.

Features	Free	Professional	Enterprise
.NET Framework Support			
Microsoft .NET Framework 1.1, 2.0, 3.5, 4.0, Silverlight, Compact Framework	✓	✓	✓
64 Bit Support	✓	✓	✓
Silverlight XAP Package		✓	✓
Obfuscation			
Symbol Renaming	✓	✓	✓
Generic Types and Methods	✓	✓	✓
XML Rule Files	✓	✓	✓
Unicode Normalization	✓	✓	✓
Overload Renaming		✓	✓
Public Symbol Obfuscation			✓
Code Protection			
Control Flow Obfuscation[1]	✓	✓	✓
Invalid Op-Codes [1]	✓	✓	✓
String Encryption [2]	✓	✓+	✓+
Protection Against Disassemblers		✓	✓
Resource Encryption [3]			✓
MSIL Encryption [4]			✓
Code Optimization			
Dead Code Removal		✓	✓
Deployment			
Strong Name Signature with SNK	✓	✓	✓
XML Map Files Generation	✓	✓	✓
Supports Resign with PFX Files		✓	✓
Assembly Embedding [3]		✓	✓
Satellite Assemblies			✓
Assembly Merging			✓
Tools Integration			
Command Line Interface	✓	✓	✓

[1] Not supported on *Compact Framework*.

[2] String encryption in the Free edition is limited to *XOR* and *Custom String* algorithms, whereas Professional and Enterprise editions supports encrypted strings hash table algorithm that provides better performance combined with an overall reduction of the obfuscated assembly's size.

[3] Not supported on *Silverlight* and Compact *Framework*.

[4] Only works with *Microsoft .NET Framework 2.0* or later.

Visual Studio Post Build Events	✔	✔	✔
MSBuild Support	✔	✔	✔
Parallel Phase Execution		✔	✔
Product Maintenance			
One Year of Product Upgrades		✔	✔

License terms and prices of Professional and Enterprise editions are available at www.babelfor.net.

Overview

The following sections will guide you through the software setup and basic understanding of command-line usage. The section Processing Phases will be a review of the obfuscation process performed by *Babel*. If you want to start using Babel right away, you can begin by reading from the Rules File section, just to get a basic understanding of how to configure the obfuscation process. Babel comes with extended command-line help syntax so that you can access the detailed help of each command line option by typing *babel –help (option name)*. The code transformations performed by Babel are grouped into phases. This manual will show you in detail what each phase does and how it can be configured using XML rules files. Following are the Visual Studio Integration and MSBuild Task sections. Those sections explain how to use Babel within *Visual Studio* and how to integrate the *Babel Obfuscator* into automated build processes. In the Advanced Topics section you will find some useful tips on improving your obfuscation and getting the most out of *Babel*. I have also included a quick reference explaining the warning messages that Babel can output (Appendix A).

If you have any comments or suggestions about the software, the website, or this help guide, please contact me at alberto.ferrazzoli@babelfor.net.

Setup

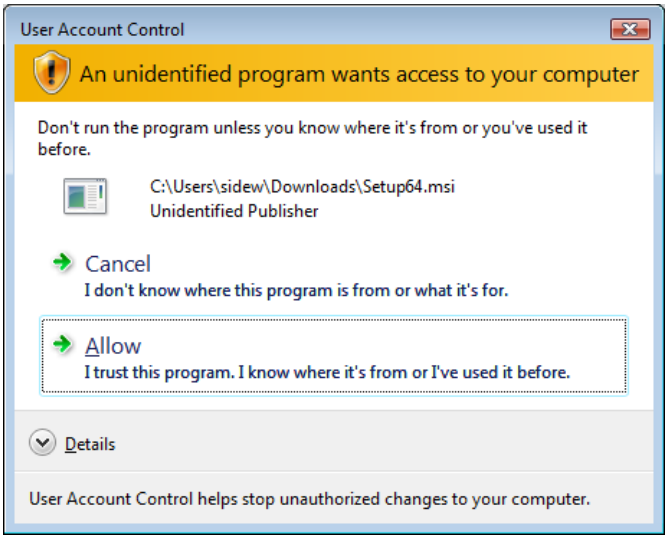
Babel Setup is available for 32 bit and 64 bit *Windows* platform as an MSI installer package. The deployment package is named *babel_x86_3.5.0.0.msi* for 32 bit and *babel_x64_3.5.0.0.msi* for 64 bit operating systems. They both provide the same compiled executables targeting all platforms. The only difference is in the location of the installed files.

The 32 bit setup will install the Babel folder under the *Program Files* folder, whereas the 64 bit version will install the Babel folder under the *Program Files 64* folder. If you run the setup package suited for 32 bit version on 64 bit operating systems the *Babel* folder will be copied under the *Program Files (x86)* directory.

NOTE

If you have a previously installed version of Babel Obfuscator, uninstall it from Control Panel before installing the new version.

To install Babel, launch the Setup MSI installer. This package will guide you through the install process. On Vista and Windows 7 operating systems you should confirm the *Allow* button in the *User Access Control* dialog during installation.



Once installed, if you want to launch *Babel* from the command shell, add the *babel.exe* executable file path to the environment user variable *PATH*:

Open Start, Control Panel, System, Advanced system settings, *Environment Variables.* In the User variables list box double click the *PATH* variable. Or if that does not exist, press the *New* button and edit in the *New User Variable Dialog* the following fields:

Variable name: PATH
Variable value: C:\Program Files\Babel

Variable value field should point to the installation path.
Now you are ready to open a command shell and start *babel*.exe.

Command Line

Babel is a command line tool, but all its functionalities are also accessible from *Visual Studio*. This section will describe how to use *Babel* from the command line.

You can start *Babel* from the command line using the following syntax:

```
babel.exe <primary assembly> [<other assemblies>...] [options]
```

The <*primary assembly*> is the target assembly that we want to obfuscate and it is mandatory.

NOTE

In the Babel command line syntax every mandatory parameter is surrounded by angle brackets <…>, whereas every optional parameter is surrounded by square brackets […].

The [<other assemblies>…] is an optional list of assemblies that will be merged with the primary assembly. The [options] are a list of command line switches that control the behavior of *Babel*. There are many options and they are grouped into several sections:

Miscellaneous

This section contains all general options that are used to configure the console output and other aspects of the obfuscation process.

--help [option] (?)

Typing *Help* without any parameters shows the main help menu with all the options available and a short description. When [option] name is specified, *Babel* will show extended help with the detailed command description.

```
C:\> babel --help stringencryption
stringencryption (nostringencryption, no-stringencryption)
usage: --[no]stringencryption [name]
  Enable ([no]disable) string encryption using an optional algorithm name
(default: enabled)

  If specified, [name] determines which algorithm babel uses to encrypt
  strings.
  Supported names are:

  hash - Compressed hash table. The strings are arranged into compressed
encrypted hash table data. This algorithm ensures also tamper protection.
  xor - Inline xor strings.

The default algorithm is hash when a valid license is present, otherwise
is the xor algorithm.
```

The short form for this option is invoked by the character “?”.

NOTE

Babel has numerous command line options, but there are two distinct kinds of options: short options and long options. Short options are a single hyphen followed by a single letter. Long options consist of two hyphens followed by a number of letters (e.g., *-t* and *--types*, respectively). Every option has a long format and sometimes an alias as well. But only certain options have an additional short format (these are typically options that are frequently used). To maintain clarity, we usually use the long form in the examples, but when describing options (if there's a short form) we'll provide the long form to improve clarity and the short form to make it easier to remember. You should use whichever one you're more comfortable with.
An option may have an alias. Aliases provide shortcuts for long options. For example *--keyfile* option has the alias *--kf*. When Babel parses long option names, it uses auto abbreviation to resolve the option.
So *--statistics* can be shortened to *--stat*. Some options can be negated. These options are like switches that can be turned on or off according to the *no* or *no-* option prefix. For example the *--types* enables type name obfuscation whereas *--notypes*, *--no-types* or *-not* disable it.

--[no]logo

This option shows the Babel copyright message. If the optional prefix [no] is specified the copyright message will not be shown.

--license

Displays available license information. If a valid license is not found, an error message will be displayed.

--verbose <n> (v)

Sets the console output verbosity level when <n> is 0 or higher. If 0 is specified, no messages are displayed during obfuscation; only warning messages and the Babel logo are displayed.

--nowarn <id>[,<id>⋯]

Disables the notification of warning messages. The *<id>[,<id>…]* parameter represents a list of warning IDs separated by a comma character. Babel will silently ignore warning numbers passed to the *--nowarn* option. All available warning IDs are listed in Appendix A.

--[no]statistics

When enabled, outputs phase processing statistics just before exiting the program.

--[no]agent

This option enables or disables *Agent Phase*. When enabled, the Agent performs static code analysis to find symbols that should not be obfuscated.

--[no]deadcode

Enable or disables the *Dead Code Removal* phase. When enabled, removes all unused methods, members, properties and events that are unused by the application.

--[no]ildasm

When enabled, Babel applies the *System.Runtime.CompilerServices.SuppressIldasmAttribute* attribute to the assembly to prevent the *Microsoft MSIL Disassembler:* ILDASM (Ildasm.exe) from disassembling the obfuscated target.

--[no]reflection

Enables or disables the protection against disassemblers.

--take <regex>

Process the XAP package assembly names that match a given regular expression. If not specified, Babel will obfuscate all assemblies that are deployed by the XAP package. These assemblies are found under the *Deployment.Parts* element in the *AppManifest.xaml* file:

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
EntryPointAssembly="OrbSl3" EntryPointType="OrbSl3.App"
RuntimeVersion="3.0.40624.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="Silverlight3App" Source="Silverlight3App.dll" />
  </Deployment.Parts>
</Deployment>
```

This option may be specified multiple times.

Examples:

```
babel package.xap --take ".*gui\.dll" --take ".*engine\.dll"
```

Will obfuscate all the DLLs file names that ends with “*gui*” or “*engine*” words. The match is done using case insensitive search.

--skip <regex>

Do not process the XAP package assembly names that match a given regular expression.
If not specified, Babel will obfuscate all assemblies that are deployed by the XAP package. These assemblies are found under the *Deployment.Parts* element in the *AppManifest.xaml* (see also --take command).

--compress <n>

Set XAP package compression level (default: 6).
XAP compression level must be between 0 and 9. If 0 is specified, the XAP package will not be compressed at all. A value of 1 maximizes speed while 9 maximizes compression. The default value is 6. High compression level may reduce XAP package size, increasing the computational time required during decompression.

Merge and Embed Assemblies

Assembly merging combines a list of assemblies into the primary assembly. This functionality is similar to that provided by *ILMerge*, a tool freely available at *Microsoft .NET Framework Developer Center*. *Babel* has its own merging algorithm that has been proven to be more efficient and less error-prone than the one provided by *ILMerge*. *Babel* merges and obfuscates all the assemblies that are listed after the primary assembly in the command line.

This section describes the command that specifically targets the merge phase.

--[no]internalize

If enabled, all the externally visible types in the merge assembly list have their visibility restricted so that only the target assembly may use them.

--[no]copyattrs

When this option is enabled, all the assembly level attributes of each input assembly are copied into the target assembly. Duplicate attributes are discarded.

--embed <assembly>

Babel can embed multiple dependency assemblies into the target assembly. The embedded assemblies are compressed and encrypted. Embedding can simplify the deployment and reduce the size of the software. It can be used instead of merge when there is no need to fully obfuscate the dependency assembly.

This option can be specified multiple times.

```
babel.exe myapp.exe --embed Library1.dll --embed Library2.dll
```

Renaming

This command line section describes the commands that can be used to configure symbol renaming.

--[no]types

If disabled, types are not renamed.

--[no]events

If disabled, events are not renamed.

--[no]methods

If disabled, methods are not renamed.

--[no]properties

If disabled, properties are not renamed.

--[no]fields

If disabled, fields are not renamed.

--[no]virtual

If disabled, virtual members such as methods, properties and events, are not renamed.

--[no]overloaded

If enabled, Babel uses the same name for two or more methods of the same type during obfuscation.

--[no]flatns

If enabled, all the types are moved into the global namespace. This flattens the namespace hierarchy and no namespace information is embedded into the obfuscated assembly.

--[no]unicode

If enabled, the names are replaced with unreadable Unicode strings. With Unicode normalization disabled all the obfuscated names are made by strings of lowercase characters taken from the Latin alphabet.

Control Flow Obfuscation

The MSIL Control Flow Obfuscation phase is processed when either *controlflow* or *invalidopcodes* are present in the command line. This phase produces a transformation of the method *MSIL* changing the code execution path so that results are difficult to understand.

--[no]controlflow

When enabled, methods' control flow is altered with the insertion of irrelevant branches.

--iliterations <n>

Set the number of iterations used in the control flow obfuscation algorithm. Setting the number of iterations to 0 disables control flow obfuscation. Increase the value *<n>* to increase the number of branch instructions inserted into each method.

--[no]invalidopcodes [mode]

Use this option to emit invalid MSIL op-codes at the beginning of every method. This will stop reflection tools like .NET Reflector to display IL method disassembly. Warning: This will make your assembly not verifiable. Normally, code that is not verifiable cannot run on x64 Operating Systems. This option should be switched off if the obfuscated assembly targets x64 platforms.

The optional parameter *[mode]* enables different IL emission configuration:

[mode] = enhanced
Insert additional invalid op-codes. This mode is not compatible with *.NET Framework 1.x*.

Encryption

MSIL encryption option enables the encryption of the method IL code. By default this option by itself does nothing if the encrypted methods are not explicitly assigned to be encrypted by means of rules or with suitable custom attributes. This explicit method designation is required because the resulting encrypted method call is much slower than the original one and the user should be aware of all the methods that will be encrypted in the target assembly.

--[no]msilencryption [regex]

If enabled, *MSIL Encryption Phase* is performed and all the methods that were designated for encryption will be processed. The methods that will be encrypted are those matching the optional regular expression or rules defined into XML files. This option can be specified multiple times.

--[no]stringencryption [name]

This option, when enabled, tells Babel to encrypt all the user strings into the target assembly. If the optional parameter *name* is provided, the encryption will be performed according the algorithm specified.

The known algorithm names are:

- *hash* – compressed hash table:
  The strings are arranged into an hash table data that is compressed and encrypted. This algorithm ensures also tamper protection.
- *xor* – inline XOR encoded strings (compatible with Silverlight 2, 3)

The default algorithm is hash when a valid license is present, otherwise is the *xor* algorithm.

--[no]resourceencryption

Enables or disables resource encryption. When enabled, all the embedded resources are compressed and encrypted.

Output Files

This section provides Babel the option to set output filenames.

--output <file>

Set the output file path for the obfuscated target. If this option is not provided, the obfuscated target will be saved into the *BabelOut* subdirectory of the original assembly folder.

--logfile <file>

Send the Babel output messages to a log file.

--mapout [file]

Set the output file name for the XML obfuscation map. If the optional parameter *[file]* is not provided, Babel will name the XML map file as the original assembly name, adding the extension *".map.xml*".

Input Files

The Input Files section contains options that are used to pass additional inputs to Babel.

--keyfile <file>

Set the strong name file used to re-sign the obfuscated assembly and localized resource DLLs.

The supported file formats are: *Strong Name Key* (.snk) and *Personal Information Exchange* (.pfx).
When a .pfx file is used, *Babel* will ask the user to enter the password during obfuscation in case it was not specified in the command line option (--keypwd <password>).

--keyname <container>

Use this option to re-sign the application if the key pair is stored in a key container. The mandatory <*container*> parameter represents the key container name used to re-sign the obfuscated assembly and localized resources DLLs.

--keypwd <password>

Specifies the password requested by a *Personal Information Exchange* (.pfx) file to re-sign the obfuscated assembly. When the password is not specified from the command line, Babel will require the user to enter the proper password during the obfuscation process.

--rules <file>

Set the input XML rule files used by Babel to configure the obfuscation process. This option can be specified multiple times. The rules files will be processed according to the order in which they are entered into the command line.

```
babel.exe myapp.exe --rules ruleset1.xml --rules ruleset2.xml
```

--mapin <file>

Set the input XML obfuscation map file that will be used to obfuscate the names of referenced symbols. This option can be specified multiple times.

Configuration File

Each option that can be passed to the command line has a default value that is stored in the *babel.exe.config* application configuration file. When the user does not explicitly specify an option at the command line, Babel will use the default value for that option. Default values are loaded from the *babel.exe.config* file. The application configuration file is a standard *.NET XML* configuration file. It contains elements which are name/value pairs for configuration information. The following table shows all possible configuration elements:

Name	Default Value	Type
ConsoleForeColorDebug	DarkGray	*System.ConsoleColor*
ConsoleForeColorError	Red	*System.ConsoleColor*
ConsoleForeColorWarning	Yellow	*System.ConsoleColor*
ConsoleForeColorInfo	White	*System.ConsoleColor*
VerboseLevel	1	*System.Int32*
ShowStatistics	True	*System.Boolean*
ShowLogo	True	*System.Boolean*
ObfuscationAgent	True	*System.Boolean*
DeadCodeElimination	False	*System.Boolean*
SuppressIldasm	True	*System.Boolean*
SuppressReflection	False	*System.Boolean*
Internalize	False	*System.Boolean*
CopyAttributes	True	*System.Boolean*
ObfuscateTypes	True	*System.Boolean*
ObfuscateMethods	True	*System.Boolean*
ObfuscateProperties	True	*System.Boolean*
ObfuscateEvents	True	*System.Boolean*
ObfuscateFields	True	*System.Boolean*
FlattenNamespaces	True	*System.Boolean*
UnicodeNormalization	True	*System.Boolean*
ObfuscateVirtualFunctions	True	*System.Boolean*
OverloadedRenaming	True	*System.Boolean*
ControlFlowObfuscation	True	*System.Boolean*
MsilEncryption	False	*System.Boolean*
ResourceEncryption	False	*System.Boolean*
IlIterations	3	*System.Int32*
EmitInvalidOpcodes	False	*System.Boolean*
EmitInvalidOpcodesMode	*empty*	*System.String*
DisableWarnings	*empty*	*System.Collection.Specialized.StringCollection*
XapCompressionLevel	6	*System.Int32*

The default value for *DisableWarnings* is an empty *System.Collection.Specialized.StringCollection*. This enables the notification of all the alarms. To add elements to this configuration section you should add an *ArrayOfString* element as follow:

```
<setting name="DisableWarnings" serializeAs="Xml">
  <value>
     <ArrayOfString xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
       <string>EM0008</string>
       <string>EM0010</string>
     </ArrayOfString>
  </value>
</setting>
```

EM008 and *EM0010* are warning IDs. The complete set of warning IDs is listed in Appendix A.

License File

The *Professional* and *Enterprise* editions come with a license file. This file needs to be copied into the *Babel* installation root folder; usually *"C:\Program Files\Babel"* and it enable the features that are associated with the edition purchased. The license file itself is an XML file digitally signed and encloses information about the product, customer details and other encrypted data that comes with each feature.
You can access your license information by typing into a DOS shell:

```
babel --license
```

If a valid license is not found the following message will be displayed:

```
Error: A valid babel license could not be found. Please contact
http://www.babelfor.net for assistance.
```

Beware that any modification to the XML license file will invalidate the license itself.
Also if your license has an expiration date, do not run *Babel* after setting back the computer system time or your license will be invalidated.

Processing Phases

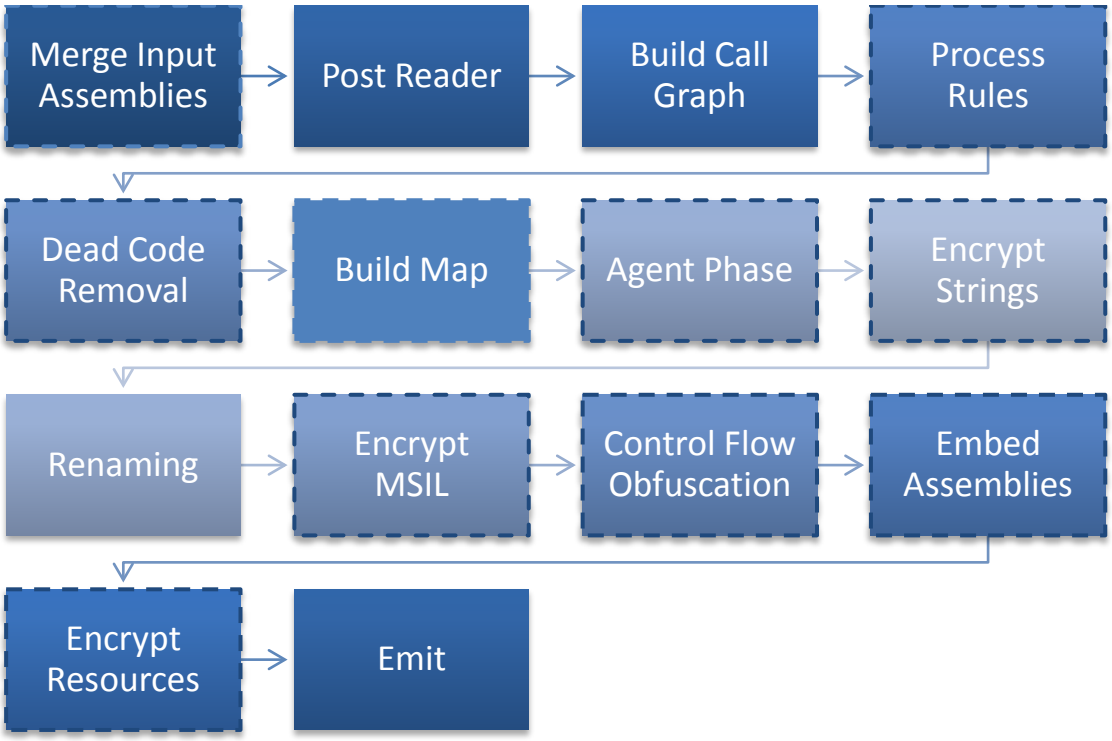
The transformations that Babel performs during the obfuscation process are grouped into phases. We may define a phase as a processing unit. Babel has a list of Phases that are processed during obfuscation. This list is not absolute and may vary according to the options passed to the command line. To configure the obfuscation process with rules it is necessary to understand how Babel organizes the phased layout. The phases that Babel arranges in its obfuscation list are outlined below in the order they are processed, according to the option passed to the command line.

Phase List	Command Line Option	Description
Merge Input Assemblies Phase	List of assemblies to process	Process XML rules files and Merge input assemblies.
Post Reader Phase		Analyze the target assembly
Build Call Graph Phase		Build the method call graph.
Process Rule Phase	--rules	Process XML rule files
Dead Code Removal	--deadcode	Remove unused members
Build Map Phase	--mapout	Build target map document
Agent Phase	--agent	Performs Agent tasks
Obfuscate Type Phase		Renaming
Encrypt String Phase	--stringencryption	Encrypt all strings

Encrypt MSIL Phase	--msilencryption	Encrypt MSIL
Control Flow Obfuscation Phase	--controlflow --invalidopcodes	Scramble MSIL code
Embed Assemblies Phase	--embed	Embed assemblies
Encrypt Resources Phase	--resourceencryption	Encrypt embedded resources
Emit Phase		Save and sign the resulting assembly

Table 1 Obfuscation Phases

Some phases are necessary to the build process (such as *Post Reader Phase* and *Emit Phase*) and they will always be present. Other phases are added only when they are needed, like *Merge Input Assemblies* and *Process Rules*. For instance, the *Process Rules* phase is added when the user specifies a rule file in the command line. The following diagram shows the flow of the Babel process. The dashed blocks represent the optional phases.



This diagram is not complete; there are phases not listed above that are used by *Bebel* as services of the main phases.

Rules File

Babel rules files are XML files that contain information used to customize the obfuscation process. The rules file consists of a list of rules defined by the XML element <*Rule*></*Rule*>. This list is enclosed by the element <*Rules*></*Rules*> that is also the root document element.

```
<Rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           version="2.0">
  <Rule name="rule 1" feature="default" exclude="false">
  </Rule>
  <Rule name="rule 2" feature="default" exclude="false">
  </Rule>
</Rules>
```

The rules element attribute **version** is used to define the current schema version and must be set to "2.0".

Rules are processed in the order they are defined in the XML Rules file. Each rule is checked against all the assembly symbols: types, methods, events, properties and fields. If a symbol name matches a rule, the obfuscation action is taken according to the value (true or false) assigned to the attribute exclude:

```
<Rule name="my rule" feature="default" exclude="false">
</Rule>
```

The Rule element has only three attributes: *name, feature, and exclude*. The *name* attribute is used to give a friendly name to the rule. The *feature* attribute defines the obfuscator feature on which the rule acts. The *name* and *exclude* attributes are mandatory. The *feature* attribute is optional, and if not specified refers to the default feature—member names obfuscation. The obfuscator feature strings are fixed. All the supported feature names are listed in the following table:

Feature	Description
all	All obfuscator features
default	Member names obfuscation
agent	Obfuscation agent
control flow	Control flow obfuscation
dead code	Dead code removal
merge	Assembly merge
msil encryption	MSIL encryption
msil encryption get stream	Declare get source stream method
renaming	Member names obfuscation
resource encryption	Embedded resources encryption
string encryption	String encryption

Table 2 Obfuscator feature table

When there are two or more rules that match a symbol name and targeting the same feature in the same XML file, the first rule declared in the file is taken into account.

NOTE

Babel is able to process multiple XML Rules files. When more than one Rules file is passed into the command line, the rules declared in each file can override an enforced rule declared in a previously processed file. So the order in which the *--rule* option is passed to the command line is important.

Any *Rule* element has other child elements: *Access, Target, Pattern, HasAttribute, Properties* and *Description*. The Pattern element is mandatory, whereas the others are optional.

<Access> - Specifies the symbol visibility for the rule. Possible values are: *All* or any combination of *Public, Protected, Internal, and Private*. If the element is not present the default value *All* is used.

<Target> - Specifies which kind of symbol should be checked. Admitted values are: *All* or any combination of *Classes, Delegates, Structures, Interfaces, Enums, Events, Methods, Properties, Fields, StaticFields* and *Resources*. If the element is not present the default value *All* is assumed.

<Pattern> - Can be any wildcard expression or regular expression. The Boolean attribute *isRegEx* states whether the pattern is a regular expression. If a regular expression is used, it's good practice to enclose the regular expression definition string in a CDATA section, ex.:

```
<Pattern isRegEx="true"><![CDATA[^Properties.*]]></Pattern>
```

<HasAttribute> - Specifies a list of fully-qualified type attributes. If the target has at least one of these attributes, the rule matches. This element may have the Boolean attribute *onEnclosingType*. If set to *false*, the attribute is checked on the target symbol itself. If the attribute is *true,* the attribute is checked on the target symbol enclosing type instead of the symbol itself.

```
<HasAttribute onEnclosingType="true">System.SerializableAttribute</HasAttribute>
```

<Properties> Defines a collection of elements used to customize the feature behavior. Each feature may support a set of property elements. For instance, the following rule sets two properties of the *msil encryption feature*: *Cache* and *MinInstructionCount*.

```
<Rule name="DataWriter" feature="msil encryption" exclude="false">
  <Target>Methods</Target>
  <Pattern>SQLUtils.DataWriter::*</Pattern>
  <Properties>
    <Cache>true</Cache>
    <MinInstructionCount>6</MinInstructionCount>
  </Properties>
  <Description>Encrypt all methods of the DataWriter class.</Description>
</Rule>
```

Follow the list of property elements supported by each feature:

Feature	Element	Type	Description
agent	TaskNameList	System.String	List of comma separated agent task names.
merge	Internalize	System.Boolean	If *true,* all types that match the rule have their visibility changed to internal.
renaming	Internalize	System.Boolean	If *true,* all types that match the rule have their visibility changed to internal.
string encryption	-	-	
control flow	IlIterations	System.Int32	The number of iterations to use in the control flow obfuscation algorithm.
	EmitIvalidOpcodes	System.Boolean	If *true*, invalid op-codes are inserted into the method definition.
msil encryption	Cache	System.Boolean	If *true,* the encrypted method is cached so that the compilation process is run only the first time the method is used.
	MinInstructionCount	System.Int32	Threshold for method minimum instruction count
	ManInstructionCount	System.Int32	Threshold for method maximum instruction count
	Source	System.String	MSIL-encrypted data source name
dead code	-	-	
resource encryption	-	-	

Table 3 Feature properties

<Description> - Any useful rule description

The following example shows the content of an XML rule files.

```
<Rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           version="2.0">
```

```
  <Rule name="Properties types" exclude="true">
    <Pattern isRegEx="false">*Properties*</Pattern>
    <Description>Do not obfuscate Properties namespace symbols.</Description>
  </Rule>
  <Rule name="Serializable types" feature="default" exclude="true">
    <Target>Fields, StaticFields</Target>
    <Pattern><![CDATA[*]]></Pattern>
    <HasAttribute  onEnclosingType="true">
       System.SerializableAttribute
    </HasAttribute>
    <Description>Do not obfuscate fields of serializable types.</Description>
  </Rule>
</Rules>
```

The first rule tells *Babel* to not obfuscate any symbols inside the *Properties* namespace or whatever symbol contains Properties in its fully-qualified name. The second rule tells Babel to not obfuscate fields of types that are declared as serializable. The *`System.SerializableAttribute`* attribute presence is checked on the field enclosing type because the XML attribute *onEnclosingType* is set to *true*.

How to Add an XML Rule File in Visual Studio Project

When *Babel* is configured to run as an *MSBuild* task in a Visual Studio Project (see Visual Studio Integration), the simplest way to add an XML Rules file is to add to the Visual Studio Project an XML file named *BabelRules.xml*. To Add an XML rules file, perform the following steps:

1) In Visual Studio Solution Explorer, right-click on the project node that you want to add the XML rules file to.
2) Select from the popup menu Add -> New Item…
3) From the Add New Item dialog, select an XML file template and enter into the Name text field: *BabelRules.xml.*
4) Fill the XML file with rule definitions.
5) Save and build the project.

Custom Attributes

Babel supports declarative obfuscation using custom attributes provided by the *.NET Framework 2.0* and attribute types declared in the *Babel.Code.dll* assembly. This section explains the usage of custom attributes to customize the obfuscation process.

System.Reflection.ObfuscateAssemblyAttribute

This assembly-level attribute instructs *Babel* to use obfuscation rules reserved to private assemblies. A private assembly is an assembly that will not be used as a library: no other software components will use the assembly. Private assemblies are fully obfuscated by *Babel* and will use obfuscation rules to rename public symbols.

The following code example shows how to mark an assembly private with the *ObfuscateAssemblyAttribute*.

```
using System;
using System.Reflection;

[assembly: ObfuscateAssembly(true)]
```

System.Reflection.ObfuscationAttribute

This attribute is used on assembly members: types, method, events, properties and fields to configure the action taken for a specific obfuscation feature. This attribute has a string *Feature* property. This property is used to map the string value to a specific obfuscator feature. The default value of *Feature* property is "*all*". *Babel* uses "*all*" to map the complete set of features and "*default*" to map the renaming feature. The complete set of feature names are the same supported by rules files and are listed in Table 2--Obfuscator feature table. The *ObfuscationAttribute* has a three other Boolean properties: *Exclude, ApplyToMembers* and *StripAfterObfuscation*. They have their default value set to *true*. The *Exclude* property indicates whether the obfuscation feature should be excluded for the associated member. The *ApplyToMembers* property specifies whether the action taken by the obfuscator for the associated type should be applied to all its members. The *StripAfterObfuscation* property specifies whether the attribute should be removed after processing.

The following code example shows how *ObfuscationAttribute* can be used to target a method for MSIL encryption:

```
using System;
using System.Reflection;

[Obfuscation(Feature = "msil encryption", Exclude = false)]
public string RandomString(int length, string characters)
{
    StringBuilder sb = new StringBuilder(length);
    for (int i = 0; i < length; i++)
    {
        int inx = _rnd.Next(characters.Length);
        sb.Append(characters[inx]);
    }
    return sb.ToString();
}
```

Babel.Code.dll Assembly

A set of custom attributes are available in the *Babel.Code.dll* assembly. This assembly is located in the Babel installation folder. The available attribute are:

Attribute type	Description
DecryptStringMethodAttribute	Used to design the custom decryption string method.
EncryptStringMethodAttribute	Used to design the custom encryption string method.
EncryptStringsAttribute	Controls the encryption of string actions on the associated method.
ObfuscateAttribute	Controls the renaming of the associated member.

Table 4 Custom attributes found in Babel.Code.dll

The *DecryptStringMethodAttribute* and *EncryptStringMethodAttribute* are used to specify the custom string decryption and encryption methods and are explained in the String Encryption section.
The *EncryptStringsAttribute* targets methods and it controls the action taken by the obfuscator for the "string encryption" feature. This attributes has an *Encrypt* boolean property. If it is set to *true* all the strings in the method will be encrypted. The *ObfuscateAttribute* targets all members and it controls action taken for the "*renaming*" feature. This attribute has a boolean *Obfuscate* property that if set to *true* tells *Babel* to rename the associated member. The *EncryptStringsAttribute* and *ObfuscateAttribute* can be replaced by the *System.Reflection.ObfuscationAttribute.*This should be the preferred attribute choice when you want to configure the action taken by the obfuscator.

All these attributes and the *Babel.Code* reference, are removed from the assembly once they are processed by the obfuscator.

Assembly Merging

With assembly merging *Babel* can embed multiple .NET assemblies into the primary assembly and obfuscate them all together. When an assembly is merged into the primary assembly, all types, MSIL code, and resources are embedded into the primary assembly. Any duplicate type will not be merged. Not all assemblies can be merged. For instance, assemblies that contain unmanaged code cannot be merged with *Babel*. Also, code that depends on the name of a merged assembly may fail when executed in the main assembly. The options available with merge are *copyattrs* and *internalize:*

The *--[no]copyattrs* option, when enabled, force Babel to copy all assembly-level attributes into the primary assembly. Any duplicate attribute will be discarded. This option is enabled by default.

The --[no]*internalize* option, when enabled, modifies the visibility of all the merged assembly types from public to internal. Because referenced types are always public, lowering their visibility to internal in the merged assembly ensures name mangling, improving the overall obfuscation statistics. This option is disabled by default, but you should enable it unless your primary assembly has to expose merged types.

Using XML Rules files you have a little more power in configuring the merge process. With Rules files you can choose types that will be merged and internalized by simply making a regular expression or a wildcard expression for the *merge* feature. For instance, the following rules merge and internalize all the types coming from the *Utilities* namespace, excluding all the other assembly types from being merged:

```
<Rule name="Utilities namespace" feature="merge" exclude="false">
  <Pattern>Utilities.*</Pattern>
  <Properties>
    <Internalize>true</Internalize>
  </Properties>
  <Description>
    Merge and internalize all type in Utilities namespace.
  </Description>
</Rule>
<Rule name="exclude all the other types" feature="merge" exclude="true">
  <Pattern>.*</Pattern>
</Rule>
```

Babel processes rules in the order in which they are declared in the rules file. So if the first rule has a mach for a given type, the second one will not be evaluated. Otherwise, if the first rule does not have a mach, the second one will—because it matches all types. And the type will not be merged.

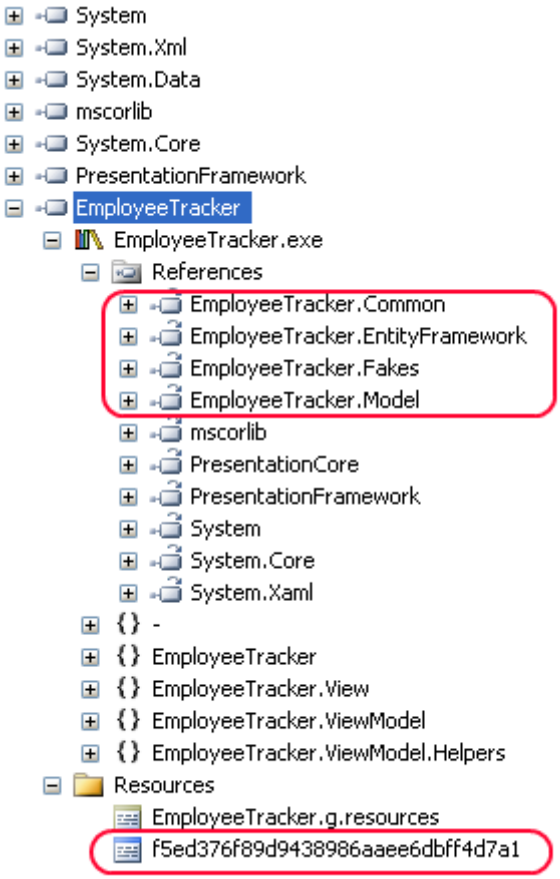
Selectively merging types with rules might break the target assembly because Babel does not perform any check on the type integrity of the resulting assembly. So it is recommended that you run the *peverify.exe* tool on the target assembly just to be sure that the rules have not left out any used type.

Assembly Embedding

Referenced assemblies can be embedded into the main application assembly using this feature. Babel can embed multiple assemblies into the main application reducing the overall size of the deployed assembly. The embedded assemblies are not obfuscated. If you need to protect the embedded assemblies you should obfuscate them first.

Embedded assemblies are stored into managed resources as compressed and encrypted data. Signing the obfuscated assembly ensures tamper protection of the encrypted data.

The following image shows a sample application *EmployerTraker.exe* after obfuscation. The referenced DLLs *EmployeeTracker.Common.dll, EmployeeTracker.EntityFramework.dll, EmployeeTracker.Fakes.dll* and *EmployeeTracker.Model.dll* were embedded into resources.



The embedded assemblies don't need to be deployed; they are loaded at runtime directly from memory.

This feature is not supported on Silverlight and Compact Framework.

Obfuscation Agent

The *Obfuscation Agent* consists of a list of tasks that *Babel* performs against the target assembly type model to determinate which symbols should not be obfuscated. The *Agent* prevents symbol names from being obfuscated according the *Common Language Runtimes* design rules. Babel is able to run a set of preconfigured tasks, and they are continuously updated in each new product release. Some of these tasks are simple—like checking fields of serizializable types. Other tasks are more complex and involve MSIL code static analysis.

The Agent works to prepare a safer obfuscation process and most of the time the final obfuscated assembly will run perfectly. Because the Agent can only perform static code analysis, it may not discover types that are controlled during execution. For example: the names of types, stored into strings that are built at runtime are opaque to the Agent. Using those strings as parameters of reflection methods impose a restriction on obfuscating the relative type or member name. Changing those type names will likely break the obfuscated assembly. For this scenario, XML rules files, combined with Agent, are the recommended solution.

Babel's Agent is enabled by default and should be left enabled whenever possible. There is generally no reason to disable the Agent entirely. But if you want, you can arrange an XML Rules file to skip a particular task from being processed. This scenario is useful when an XML rule is overridden by an Agent task outcome. Agent tasks run after the Rules processing phase, so any custom XML Rule enforced on a member will be overridden by the Agent task. The only way to disable agent rules is to define an XML Rule to switch off the Agent task on that particular member.

For instance, suppose that you want to obfuscate a public type named *RedBlackTree*. So you prepare a rule to force Babel to rename the type:

```
<Rule name="RedBlackTree" feature="renaming" exclude="false">
  <Target>Classes</Target>
  <Pattern>*.RedBlackTree</Pattern>
  <Description>Rename RedBlackTree class.</Description>
</Rule>
```

Then you discover that the type was not renamed because the Agent found that the type is serializable, and serializable types are not renamed. But suppose you don't care about serialization and you want that type to be obfuscated. You can get around this in two different ways. One way is to tell Babel to turn off Agent analysis with the option *–noagent*. But this is overkill for our purpose, and the application process might be broken after the obfuscation because the Agent has not run. The better way, however, is to add a second rule that forces the Agent to ignore the RedBlackTree type when the serialization task is running.

```
<Rule name="RedBlackTree agent" feature="agent" exclude="true">
  <Pattern>.*RedBlackTree</Pattern>
  <Properties>
    <TaskNameList>Serializable types</TaskNameList>
  </Properties>
  <Description>Ignore Serializable types tasks for RedBlackTree.</Description>
</Rule>
```

In this rule we have two elements that are meant to be explained.

- The **feature** is set to agent to tell the rule processor that we want the rule to be applied to the agent.
- The presence of the element **Properties** with the property **TaskNameList** that contains the name of the agent task "`Serializable types`". This tells the rule processor that we want to exclude only the analysis carried on serializable types.

List of available Agent task names:

Task name string	Description
Reflected enums	Ignore enum types that are consumed by reflection methods
Serializable types	Do not rename serializable types
Reflected strings	Discover symbol name strings consumed by reflection methods
Reflected types	Do not rename types consumed by reflection methods
Type consumed by attributes	Do not rename types consumed by attributes
Exposed attributes	Ignore symbols that expose specific attributes
XAML analyzer	Parse XAML resources to discover types that should not be obfuscated

Table 5 Agent task names

Renaming

The Renaming phase performs assembly symbol name obfuscation. Types, methods, properties, events, fields, namespaces and method parameter names are changed into a combination of alphabet character or unprintable Unicode symbols. The original symbol name is lost and is impossible to recover the meaning of the obfuscated member. This makes code understanding and reverse engineering extremely difficult.

You can choose from two different renaming conventions: alphabet characters or unprintable Unicode symbols. Both minimize the number of character symbol used, reducing the overall metadata size and load time. Babel uses the Unicode renaming schema as default, but you can switch to alphabet character set by specifying the option *–nounicode* in the command line.

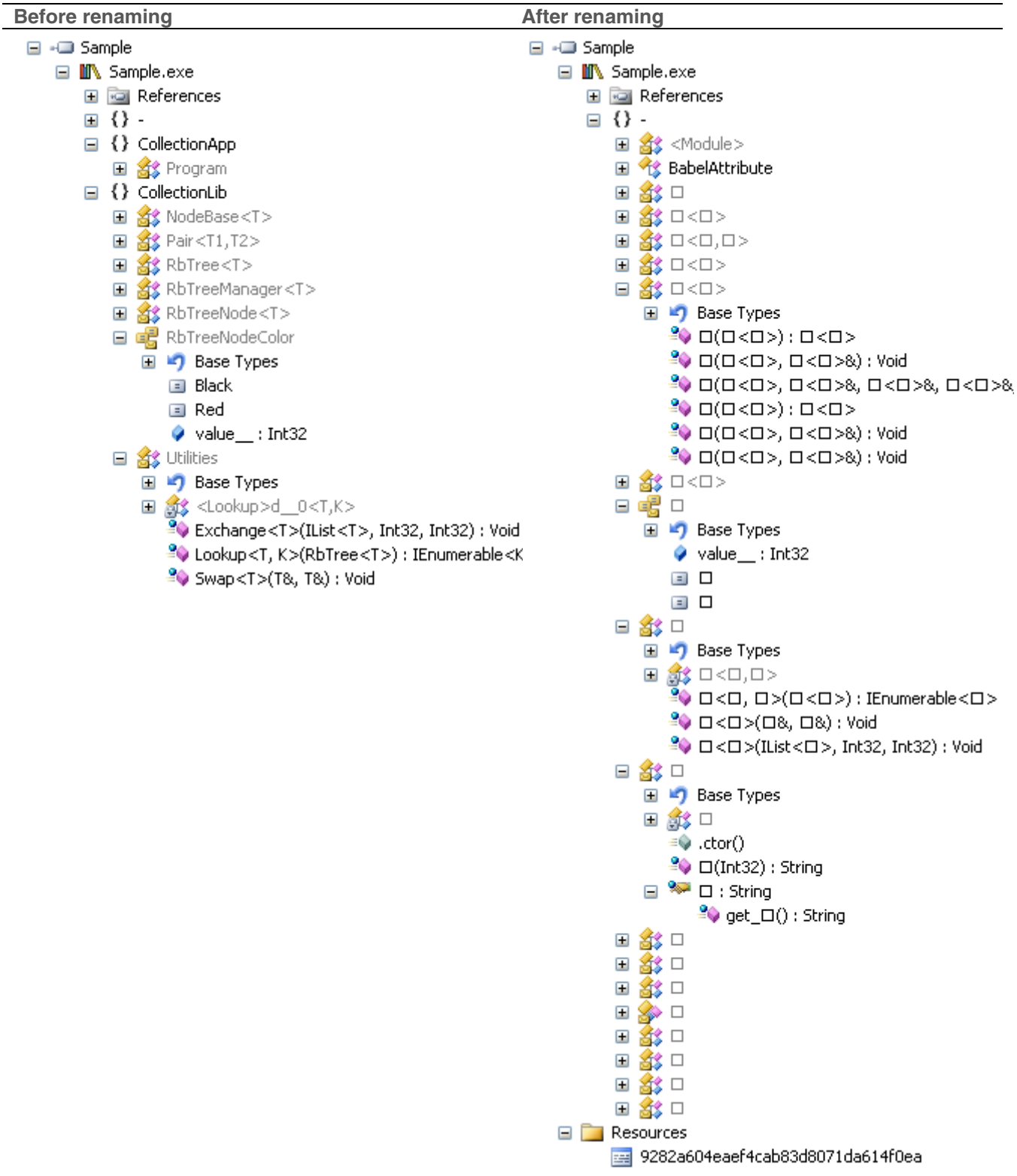
The Renaming process has several options available for configuration. You might consider disabling renaming by using the switches --notypes, --nomethods, --noevents, --noproperties and –nofiedls, or their equivalent short form: -notmepf. This is particularly useful when debugging obfuscation issues. Also the –novirtual option will disable the renaming of virtual symbols (methods, properties, events), which can be used for the same purpose.

The *--flatns* option controls the namespace renaming. When this option is enabled, all the root namespace types are collapsed into the global namespace.

NOTE

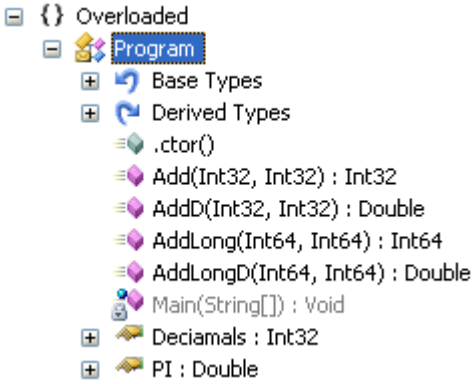
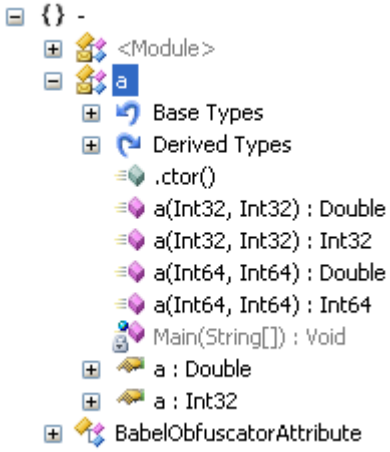
Externally-visible symbols will not be renamed by Babel unless you specify a custom rule to force that. To increase the overall number of members obfuscated, considers lowering the visibility of your public types to internal.

The following picture shows an assembly before and after name obfuscation using Unicode normalization.



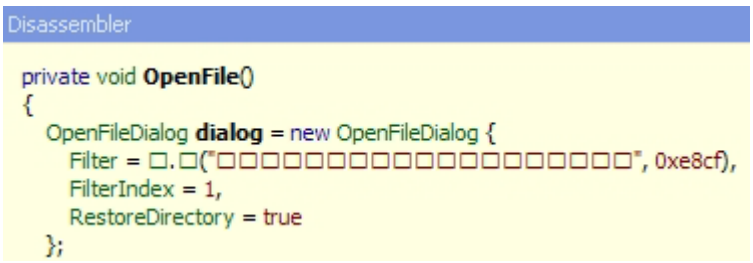
Overloaded Renaming

With overloaded renaming, the same name is used to rename methods with different signatures as long as it is allowed by .NET Framework design rules. This makes it even more difficult for a hacker to reverse engineer the code. Babel also renames overloads when only the method return type differs, making it impossible to entirely decompile the code to high-level languages like C# and VB.NET, in which overloading the return type is not permitted.

Source	Overloaded Renaming
	

String Encryption

Babel can encrypt so-called *User Strings* embedded into a managed assembly. User Strings are all strings found in the heap blob that are referenced in the user code. The strings are kept in Unicode encoding.
Babel has different methods (algorithms) of encrypting strings: *xor* and *hash*. Each algorithm can be specified in the command line by using the option –string. If the user does not specify the encryption method, Babel will choose the hash algorithm if a *Professional* or *Enterprise* license is available otherwise it will chose the xor algorithm. The xor algorithm is the simplest one and consists of xor-ed inlined string characters with a random integer key. This method is not tamper proof.
The *xor* method encryption is shown in the following screenshots. The image on the left shows the original code disassembly with a string referenced by the instantiation of the *OpenFileDialog* class, while the image on the right shows the same code after xor string encryption.

Code	Xor string encryption
	

The *hash* algorithm is based on hash tables addressed by integer keys. This algorithm performs compression and encrypted of string data to reduce the overall file size of the obfuscated assembly. This algorithm also ensures tamper-proof protection.

The following picture shows *hash* string encryption algorithm code disassembly:

Code	Hash string encryption
	

Babel also supports custom string encryption. With custom string encryption the user must provide in the target assembly two methods: one for encrypting strings named *EncryptString,* and one named *DecryptString*. The prototype for these two methods is the same: they both take a string as parameter and returns its encrypted or decrypted string value:

```
/// <summary>
/// Encrypt an user string.
/// </summary>
/// <param name="text">The user string to encrypt.</param>
/// <returns>The encrypted user string.</returns>
internal static string EncryptString(string text)
{
    // Encrypt text string and return the encrypted string object.
    return ...;
}

/// <summary>
/// Decrypt an encrypted user string.
/// </summary>
/// <param name="text">The encrypted string.</param>
/// <returns>The decrypted user string.</returns>
internal static string DecryptString(string text)
{
    // Decrypt text string and return the decrypted string object.
    return ...;
}
```

Babel will search for these two methods, and use them when encrypting strings. At the end of the string encryption phase, Babel will remove the *EncryptString* methods because they're not necessary at runtime. Another way to specify custom string encryption methods is to add the attributes `EncryptStringMethodAttribute` and `DecryptStringMethodAttribute` for the encrypt and decrypt methods respectively. When these two attributes are specified, the EncryptString and DecryptString methods can have any user-defined name:

```
using Babel.Code;

[EncryptStringMethod]
internal static string MyEncryptString(string text)
{
}

[DecryptStringMethod]
internal static string MyDecryptString(string text)
{
}
```

These two attribute types are defined in the *Babel.Code.dll* assembly that must be referenced into the project References in order to be used.

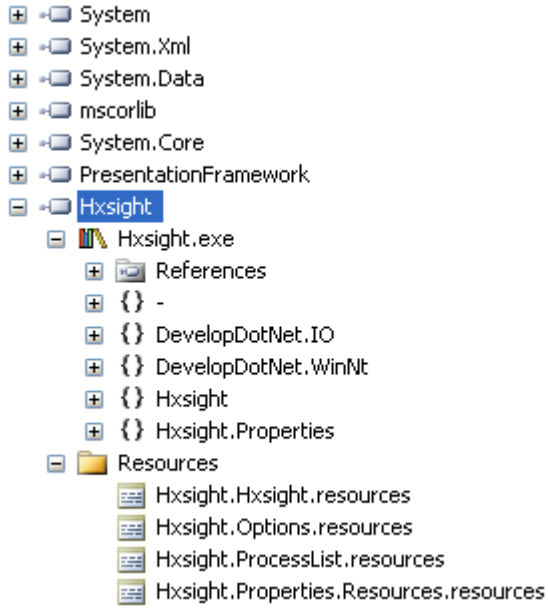
Follow the encryption algorithm compatibility table.

Algorithm	.NET Framework 1.1	.NET Framework 2.0	.NET Framework 3.5	.NET Framework 4.0	Silverlight	Compact Framework
xor	✔	✔	✔	✔	✔	✔
hash	✔	✔	✔	✔	✔	✔
custom	✔	✔	✔	✔		✔

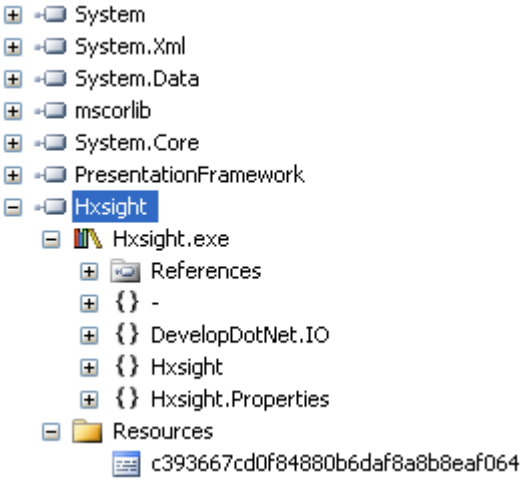
Resource Encryption

Babel can compress and encrypt embedded resources to protect your resources and also to reduce the overall assembly size. The encrypted resources are loaded at runtime when they are eventually needed. The following pictures show an assembly with embedded resource before and after resource encryption.

Before resource encryption



After resource encryption



Resource encryption is not supported on Silverlight and Compact Framework.

Control Flow Obfuscation

Control Flow obfuscation is enabled with the command line option *–controlflow*. When enabled, Babel changes the execution path of a method inserting a number of irrelevant branches so that, although the behavior of the method is not changed, it is very difficult to analyze after decompilation. Consider the following example that shows a method before and after control flow obfuscation.

Before control flow obfuscation

Disassembler

```
private string ConvertToHexString(byte[] data)
{
   StringBuilder builder = new StringBuilder();
   if (null != data)
   {
      for (int i = 0; i < data.Length; i++)
      {
         builder.Append(data[i].ToString("X2"));
      }
   }
   return builder.ToString();
}
```

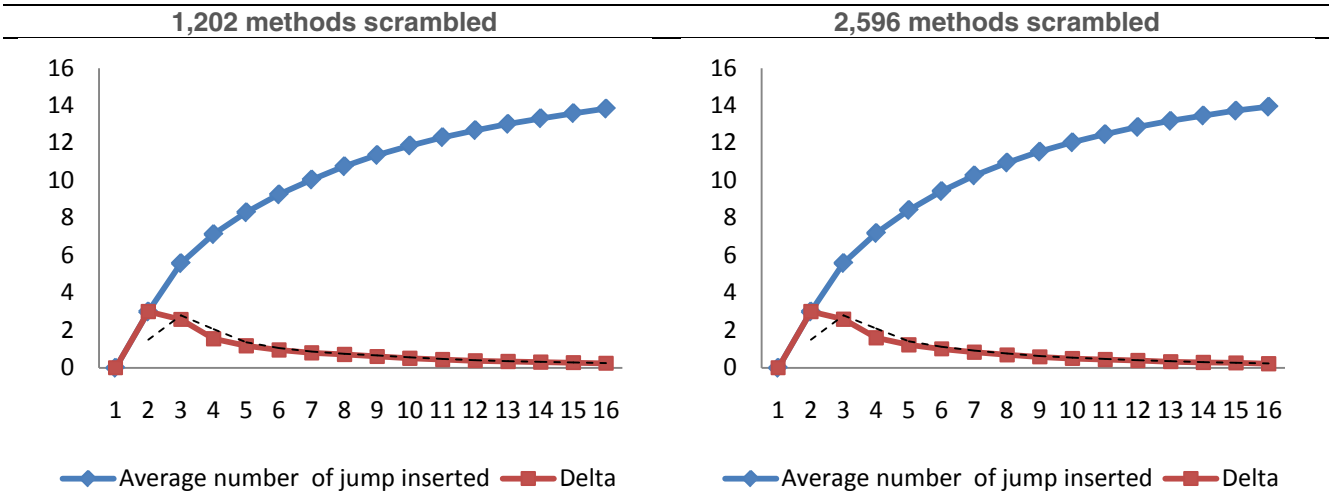
Note: after control flow obfuscation is performed, it is no longer possible to reconstruct the original if-else and loop clause for disassembly, since that C# code listing is no longer available and only the IL listing can be displayed.

After control flow obfuscation

Disassembler

```
.method private hidebysig instance string ConvertToHexS
{
   .maxstack 7
   .locals init (
      [0] class [mscorlib]System.Text.StringBuilder builder,
      [1] bool flag,
      [2] int32 num,
      [3] string str)
   L_0000: nop
   L_0001: newobj instance void [mscorlib]System.Text.Strin
   L_0006: stloc.0
   L_0007: br.s L_004d
   L_0009: ldarg.1
   L_000a: br.s L_0013
   L_000c: stloc.1
   L_000d: ldloc.1
   L_000e: brtrue.s L_004a
   L_0010: nop
   L_0011: br.s L_0017
   L_0013: ceq
   L_0015: br.s L_000c
   L_0017: ldc.i4.0
   L_0018: stloc.2
   L_0019: br.s L_003f
   L_001b: nop
   L_001c: br.s L_003c
   L_001e: ldarg.1
   L_001f: ldloc.2
   L_0020: ldelema uint8
   L_0025: ldstr "X2"
```

The number of jumps inserted by Babel can be changed in a certain amount changing the number of iterations performed by the control flow obfuscation algorithm. This parameter is configured by the command line option switch *–iliteration* <*n*>. Increasing the <*n*> value may result in an increase of the number of irrelevant branches inserted. The <*n*> value does have an upper limit where the control flow cannot be altered anymore by the algorithm. This limit depends on the method control flow structure and can be different according to the particular method considered. In this case, increasing <*n*> does not result in an increased number of jumps. The following graphs show an average number of jumps inserted (vertical axes) varying with the number of iterations <n> imposed (horizontal axes) for two different assemblies. The first assembly with 1,202 methods processed is shown on the left, whereas the second assembly with 2,596 methods processed is on the right. The trend dash lines show that the optimum iteration value is about 3, which is also the default value.



It is not recommended to increase the <n> parameter over 6, as higher values will scramble the code more, causing the overall execution time to increase.

Scrambling the control flow makes the obfuscated target impossible for IL verification, since maximum stack size for a method cannot be evaluated with a single-forward scan of the method. This is noticeable when you run the obfuscated target through the *peverify.exe Microsoft SDK* tool.

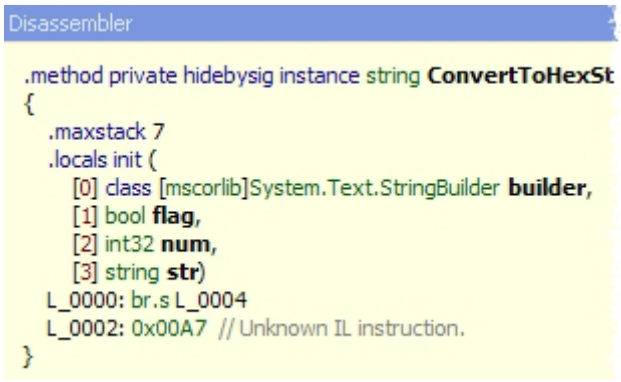
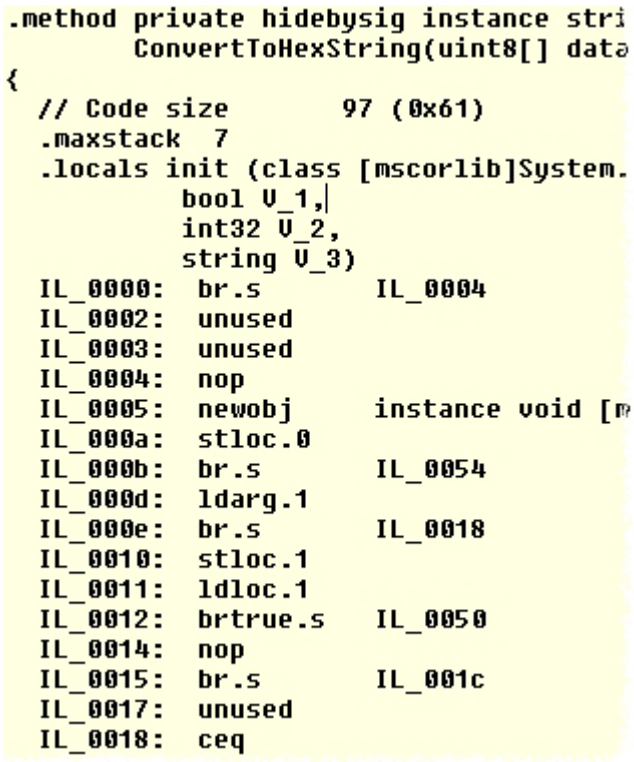
```
[IL]: Error: […][offset 0x00000050] Stack height at all points must be
determinable in a single forward scan of IL.
```

This is not a problem because the runtime will always execute the control flow obfuscated method.

Invalid Op-Codes

Invalid op-codes are byte codes not recognized by the runtime as valid MSIL instructions that *Babel* inserts in each method to prevent the disassembler from decompiling the method. Invalid op-codes are not managed properly by disassemblers like *.NET Reflector*, while the ILDASM disassembler can list the MSIL even with the presence of invalid bytes. Therefore it is recommended to disable the ILDASM tool with the option *–noildasm* whenever possible.

The following table shows, on the left, the *.NET Reflector* method IL listing after obfuscation with invalid op-codes. On the right side the same method is represented by ILDASM disassembler.

.NET Reflector IL code listing	ILDASM code listing
	

The unused instructions at offset 2, 3 and 17 that *ILDASM* output in the code disassembly reflect the inserted invalid op-codes.

The number of iteration used in the control flow algorithm as well as the invalid op-codes emission can be configured using XML rules. There are two properties available in the control flow feature that are applied on each method that match the rule: *IlIterations* and *EmitInvalidOpcodes*. The first is a positive integer and set to the number of iterations for control flow algorithm. The latter is a Boolean and it is for the emission of invalid op-codes.

Invalid op-code emission should not be performed if the obfuscated assembly targets x64 operating systems (see Obfuscating x64 Assemblies).

MSIL Encryption

The MSIL encryption provided by Babel is a completely managed solution. This means that the encrypted methods are not replaced by native code targeting a particular platform. The managed method encryption ensures that the cross platform nature of the .NET Framework is not compromised. Moreover, the runtime compilation of encrypted methods allows the Just-In-Time (JIT) compiler to optimize code for the target CPU. When a method is encrypted, its code is replaced by a call to a stub method that decrypts and compiles a dynamic method (*System.Reflection.Emit.DynamicMethod*) at runtime. The original MSIL code is compressed and encrypted into the assembly resources. Babel ensures tamper protection on the encrypted method resource when the original assembly is strong-name signed.

When a method is encrypted, its MSIL code is replaced by a call to a stub method that ensures decryption, runtime compilation and execution of the dynamic method. The process of decryption, compilation and execution of the dynamic method is much slower than the original method execution, so the user must choose carefully the list of methods to encrypt. The user can select a method to encrypt with XML rules files or by applying on the method the *System.Reflection.ObfuscationAttribute* attribute. Rules files are more flexible and allow the user to specify encryption options that are not supported by the custom attribute.

For instance, suppose you have a type named Rgb that encapsulates RGB color components, and you want to encrypt all its methods. If the Rgb class contains few methods you can apply to each method the *System.Reflection,ObfuscationAttribute* as follow:

```
namespace Utilities
{
    public class Rgb
    {
        byte _r; byte _g; byte _b;

        public Rgb(byte r, byte g, byte b)
        {
            _r = r; _g = g; _b = b;
        }

        public UInt32 Value
        {
            [Obfuscation(Feature = "msil encryption", Exclude = false)]
            get
            {
                return ToUint32();
            }
        }

        [Obfuscation(Feature = "msil encryption", Exclude = false)]
        public UInt32 ToUint32()
        {
            return (UInt32)((_r << 16) | (_g << 8) | _b);
        }

        [Obfuscation(Feature = "msil encryption", Exclude = false)]
        public static UInt32 ToUint32(Rgb rgb)
        {
            return rgb.ToUint32();
        }
    }
}
```

When the number of methods on the Rgb type becomes significantly large it is better to define a custom XML rule:

```
<Rule name="Rgb" feature="msil encryption" exclude="false">
  <Target>Methods</Target>
  <Pattern>Utilities.Rgb::*</Pattern>
  <Description>Encrypt all methods of Rgb class.</Description>
</Rule>
```

The above rule defined will encrypt all the methods of Rgb class.

Encrypting too many methods may results in slowing down excessively the application during execution. In this scenario the developer can tune the number of method that will be encrypted using custom properties enabled for the specific feature “*msil encryption*”.
These custom properties are:

- MinInstructionCount
- MaxInstructionCount
- Cache
- Source

The *MinInstructionCount* element sets the lower limit for the number of MSIL instuctions that a method must have to be encrypted. Conversely, the *MaxInstructionCount* element sets the upper limit for the number of MSIL instuctions that a method must have to be encrypted. The *Cache* element contains a boolean value, and when enabled, all the *DynamicMethod* attributes that match the rule are cached into memory so that the compilation is performed only the first time the method is called. This optimization is enabled by default. The user may decide to disable caching to ensure that the method is discarded after being used in order to free memory resources.

```
<Rule name="Rgb" feature="msil encryption" exclude="false">
  <Target>Methods</Target>
  <Pattern>Utilities.Rgb::*</Pattern>
  <Properties>
    <Cache>true</Cache>
    <MinInstructionCount>5</MinInstructionCount>
    <MaxInstructionCount>50</MaxInstructionCount>
  </Properties>
  <Description>Encrypt methods of Rgb class.</Description>
</Rule>
```

The *Source* element contains a string value and its use will be described in the Customize MSIL Encryption section.

The following screenshot illustrates a method before MSIL encryption:

Disassembler

```
private void InvalidateTriangle(int index)
{
  if ((index >= 0) && (index < this.vals.Length))
  {
    Rectangle rectangle;
    int num = this.ValueToPosition(this.vals[index]);
    switch (this.orientation)
    {
      case Orientation.Horizontal:
        rectangle = new Rectangle(num - 3, 0, 7, base.Height);
        break;

      case Orientation.Vertical:
        rectangle = new Rectangle(0, num - 3, base.Width, 7);
        break;

      default:
        throw new InvalidEnumArgumentException();
    }
    base.Invalidate(rectangle, true);
  }
}
```

And after MSIL encryption transformation:

Disassembler

```
private void □(int □)
{
  □.□("□", new object[] { this, □ });
}
```

Note that all the code inside the *InvalidateTriangle* method has been replaced by a single call to an internal method that performs decryption and compilation of a dynamic method equivalent to the original method.

This encryption method has some limitations and not all methods can be encrypted. The methods that do not support MSIL encryption are:

- Instance constructor
- Generic methods
- Methods that uses generic non-instance types
- Methods with ref or out parameters

If the user tries to obfuscate such methods, Babel will raise a warning and the method will not be encrypted. These limitations pose a restriction on the method that can be encrypted, but because method encryption is not intended to be used extensively on the target assembly this limitation is not very important. The developer should encrypt only highly sensitive methods. A few examples of methods you may want to encrypt:

- License file parsing
- Activation key validation
- Proprietary algorithm

Most of the time, these methods can be arranged so that Babel can successfully encrypt the method.

MSIL Encryption is supported only on .NET Framework 2.0 and later, and is not supported on Silverlight and Compact Framework.

Visual Studio Integration

This section will describe how you can use Babel Obfuscator with common build tools like *Visual Studio* and *MSBuild*. Visual Studio integration offers two different ways of obfuscating project assemblies. The first one involves the editing of visual studio project files and it invokes Babel's *MSBuild* task. The latter consists in launching *babel.exe* from the post-build event Project Designer page.

Project File

To integrate Babel within Visual Studio (.*proj*) file, open the project file and insert an *Import* element in the *Babel.Build.targets* file. This file is located under the *MSBuild* folder inside the *Babel* installation directory (usually *C:\Program Files\Babel*).

Follow the steps required to modify the Visual Studio (.*proj) file:

1) In *Visual Studio* Solution Explorer, select the project node, right-click, and from the context popup menu choose the *Unload Project* command. When the project is unloaded, its icon changes to a folder and the word '(unavailable)' appears next to it.
2) Right click the unloaded project, and from context popup menu choose *Edit*.
3) In the XML project file insert after the line:

```
<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
```

The following Import declaration:

```
<Import Project="<full path to Babel.Build.targets file>" />
```

For example:

```
<Import Project="C:\Program Files\Babel\MSBuild\Babel.Build.targets" />
```

4) Save and reload the project.

Rebuild the project and look at the *Visual Studio* Output panel to see Babel console output. The obfuscated assembly is saved to *.\BabelOut* sub-folder relative to the target directory.

The *Babel.Build.targets* file contains the declaration of the *Babel MSBuild* task. This task starts babel.exe and passes into the command line the option specified in the xml task definition. The default binding of Visual Studio project properties with the Babel task options is specified in the *Babel.Build.targets* file. These properties can be overridden in your (.*proj)* file to customize the Babel task behavior. For instance, in *Babel.Build.targets* the property *UnicodeNormalization* is set to *true* to enable Unicode normalization by default:

```
<PropertyGroup>
  <UnicodeNormalization>true</UnicodeNormalization>
</PropertyGroup>
```

If you want to disable Unicode normalization, insert into the "*.*proj"* file after the *Babel* import declaration:

```
<Import Project="C:\Program Files\Babel\MSBuild\Babel.Build.targets" />
```

The following lines:

```
<PropertyGroup>
  <UnicodeNormalization>false</UnicodeNormalization>
</PropertyGroup>
```

Other properties are available to override. The complete set of available task properties is listed in the Babel Task section.

There is a property that can be used to switch *Babel* obfuscation ON or OFF in particular build configurations. This is the *EnableObfuscation* property, and it is useful if you want to enable obfuscation in *Release* builds, but disable in *Debug* builds:

For instance you can add the following lines to your "*.*proj*" file:

```
<Choose>
  <When Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <PropertyGroup>
      <EnableObfuscation>false</EnableObfuscation>
    </PropertyGroup>
  </When>
  <When Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <PropertyGroup>
      <EnableObfuscation>true</EnableObfuscation>
      <ILIterations>3</ILIterations>
      <StringEncryption>true</StringEncryption>
    </PropertyGroup>
  </When>
</Choose>
```

When the *Debug* build is selected, the *Babel* task is disabled so that the *babel.exe* is not executed at the end of the build process. Whereas the *Babel* task is enabled when *Release* build is active.

Post Build Event

If you don't want to edit Visual Studio projects, you can always obfuscate your assemblies from within Visual Studio by launching *babel.exe* in a post build event. Just open the Build Events page of the Project Designer to specify the babel.exe post build command:

```
babel.exe "$(OutDir)$(TargetFileName)" -v1 --keyfile "$(ProjectDir)StrongName.snk"
--rules "$(ProjectDir)babelRules.xml"
```

To successfully run the event command, babel.exe must be available in your system PATH. The command-line syntax can include any build macro.

MSBuild Task

Babel comes with support for the *MSBuild* tool by means of a custom task that performs the *Babel* obfuscation launching the babel.exe command with options read from task properties. The Babel *MsBuild* task is defined in the assembly *Babel.Build.dll* located in the *MsBuild* folder under the program installation directory (usually: *C:\Program Files*) and also under the system *Global Assembly Cache*.

Babel Task

Babel task can be added to any *MSBuild* project by inserting the appropriate *UsingTask* element into the *MsBuild* project xml file:

```
<UsingTask TaskName="Babel" AssemblyName="Babel.Build, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=138d17b5bd621ab7" />
```

Consider the following very simple *C#* project made by one executable assembly that references one DLL library. We use this sample to describe how to make an *MSBuild* project that uses the Babel task. The main assembly *HelloWorld.exe* is a console application that makes a call to a static method *Hello.ToConsole()* defined in the *HelloLib.dll* assembly:

Main.cs

```
using System;
using HelloLib;

namespace HelloApp
{
    class Program
    {
        public static void Main(string[] args)
        {
            Hello.ToConsole();
        }
    }
}
```

HelloLib.cs

```
using System;

namespace HelloLib
{
    public class Hello
    {
        public static void ToConsole()
        {
            Console.WriteLine("Hello");
        }
    }
}
```

First we need to create an *MSBuild* project file: *Hello.proj* to build the *Main.cs* and *HelloLib.cs* source files into their respective assemblies and include the *UsingTask* element to reference the *Babel* task.

```
<?xml version="1.0" encoding="UTF-8"?>
<Project
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

  <UsingTask TaskName="Babel" AssemblyName="Babel.Build, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=138d17b5bd621ab7"/>

</Project>
```

To tell *MsBuild* to compile the *HelloLib.cs* file into HelloLib.dll assembly, we define a Task that calls the C# compiler (*CSC*) on *HelloLib.cs* and specify the file format *library* for the output file using the *TargetType* attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<Project
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

  <UsingTask TaskName="Babel" AssemblyName="Babel.Build, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=138d17b5bd621ab7"/>

  <ItemGroup>
    <CSLibFile Include="HelloLib.cs"/>
  </ItemGroup>

  <Target Name="HelloLib">
    <Message Text="Building $(TargetFile)..."/>
    <CSC Sources="@(CSLibFile)"
         WarningLevel="$(WarningLevel)"
         TargetType="library">
    </CSC>
  </Target>
</Project>
```

Now we need to define another Target that compiles the file *Main.cs* and builds the *HelloWorld.exe* application using as reference the *HelloLib.dll* library:

```
<?xml version="1.0" encoding="UTF-8"?>
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

  <UsingTask TaskName="Babel" AssemblyName="Babel.Build, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=138d17b5bd621ab7"/>

  <PropertyGroup>
    <AppName>HelloWorld</AppName>
    <LibName>HelloLib</LibName>
  </PropertyGroup>

  <ItemGroup>
    <CSLibFile Include="HelloLib.cs"/>
  </ItemGroup>

  <ItemGroup>
    <CSFile Include="Main.cs"/>
  </ItemGroup>

  <Target Name="HelloLib">
    <Message Text="Building $(LibName)..."/>
    <CSC Sources="@(CSLibFile)"
         WarningLevel="$(WarningLevel)"
         TargetType="library">
    </CSC>
  </Target>

  <Target Name="Build" DependsOnTargets="HelloLib">
    <Message Text="Building $(Appname)..."/>
    <CSC Sources="@(CSFile)"
         References="$(LibName).dll"
         WarningLevel="$(WarningLevel)"
         OutputAssembly="$(AppName).exe">
    </CSC>
  </Target>
</Project>
```

Because the project file contains multiple targets we need to specify the *DefaultTargets* attribute into the Project element so that *MSBuild* knows to execute the *Build* target first. Note that the *Build* target depends on the *HelloLib* target because the referenced assembly needs to be available when the *Main.cs* file is compiled. We can build the *HelloWorld.exe* application by entering into the command shell the command:

```
msbuild.exe Hello.proj
```

We now have the *HelloWorld.exe* application, but we did not use the *Babel* task yet. Suppose that we want to obfuscate only the *HelloWorld.exe* assembly. This can be easily done by adding to the *Build* target the *Babel* task declaration:

```
<Target Name="Build" DependsOnTargets="HelloLib">
  <Message Text="Building $(Appname)..."/>
  <CSC Sources="@(CSFile)"
       References="$(LibName).dll"
       WarningLevel="$(WarningLevel)"
       OutputAssembly="$(AppName).exe">
  </CSC>

  <Babel InputFile="$(AppName).exe"/>

</Target>
```

The only task attribute that we need to specify is the *InputFile* attribute. This attribute is mandatory and is equivalent to specifying the primary assembly source into the *babel.exe* command line. The task so defined will start *babel.exe* to obfuscate the *HelloWorld.exe* primary assembly using the default options. The obfuscated assembly will be saved into the *BabelOut* folder.

The *Babel* task is able to locate the *babel.exe* tool under the system "*Program Files"* folder when installed into the standard *Babel* directory. If the Babel setup was performed to install the program under a directory with a different name, the Babel task will not be able to find the *babel.exe* executable and the build process will terminate with an error. In this scenario we need to specify the *Babel* installation directory as input to the *Babel* task by using the attribute *BabelDirectory:*

```
<Babel BabelDirectory="C:\Program Files\Babel Obfuscator"
InputFile="$(AppName).exe"/>
```

You can add more attributes to customize the Babel behavior. For instance, if you want to save the obfuscated assembly with a different name you can add the *OutputFile* attribute with the name of the obfuscated executable:

```
<Babel InputFile="$(AppName).exe"
       OutputFile="$(AppName)_babel.exe"/>
```

The obfuscated assembly will be saved in the same folder as the primary source with the name: *HelloWorld_babel.exe*. You can also change the encryption string algorithm used when Babel encrypt strings by specifying the *StringEncryptionAlgorithm* attribute:

```
<Babel InputFile="$(AppName).exe"
       StringEncryptionAlgorithm="hash"
       OutputFile="$(AppName)_babel.exe"/>
```

You can also instruct the *Babel* task to merge the referenced assembly *HelloLib.dll* into the primary assembly by adding the *MergeFiles* attribute:

```
<Babel InputFile="$(AppName).exe"
       MergeFiles="$(LibName).dll"
       Internalize="true"
       StringEncryptionAlgorithm="hash"
       OutputFile="$(AppName)_babel.exe"/>
```

The *Internalize* attribute changes the visibility of *Hello* class so that it will be visible only to types defined internally to the merged assembly, making the *Hello* class name available to be obfuscated. If you wanted to merge multiple libraries into the primary assembly you need to specify the names of the libraries separated by a semicolon:

```
MergeFiles="$(LibName1).dll;$(LibName2).dll"
```

Now suppose that you wanted to encrypt all the methods defined in the HelloLib.dll assembly. Usually to do this you would need to define a rules XML file to target the *msil encryption* feature and pass to the *babel.exe* command line the *msilencrypt* option. With a *Babel* task you can do the same by adding two attributes: the first, *RulesFiles,* defines a list of XML rules files to use as input. The second, *MsilEncryption,* tells *babel.exe* to run the MSIL encryption phase.

```
<Babel InputFile="$(AppName).exe"
       MergeFiles="$(LibName).dll"
       Internalize="true"
       StringEncryptionAlgorithm="hash"
       RulesFiles="babelRules.xml"
       MsilEncryption="true"
       OutputFile="$(AppName)_babel.exe"/>
```

The *babelRules.xml* file is defined as follow:

```
<?xml version="1.0" encoding="utf-8"?>
<Rules version="2.0">
  <Rule name="encrypt" feature="msil encryption" exclude="false">
    <Target>Methods</Target>
    <Pattern>HelloLib.*</Pattern>
    <Properties>
      <Cache>true</Cache>
    </Properties>
    <Description>Encrypt HelloLib methods.</Description>
  </Rule>
</Rules>
```

There are many attributes available to customize the obfuscation process under *MSBuild.* The following table shows all the known *Babel* task attributes and the related properties defined in the *Babel.Build.targets* file. These properties are used by *MSBuild* only with Visual Studio integration.

Task Attribute	***Babel.Build.targets***	Type	Description
BabelDirectory	BabelDirectory	String	Full path to babel.exe tool
-	EnableObfuscation	Boolean	Global switch to enable or disable the obfuscation process
InputFile	*TargetPath*[5]	String	The path of the main source assembly (mandatory)
OutputFile	BabelOutputFile	String	Obfuscated target file path
ShowLogo	ShowLogo	Boolean	Outputs Babel copyright information
ShowStatistics	ShowStatistics	Boolean	Whether you want Babel to show obfuscation statistics
VerboseLevel	VerboseLevel	Int32	Console output verbosity
EnableObfuscationAgent	EnableObfuscationAgent	Boolean	Whether to enable agent task processing
-	BabelRulesFileName	String	BabelRules.xml

[5] The *Italic* boldface properties listed in the table are those reserved by *Visual Studio MSBuild* projects.

Task Attribute	*Babel.Build.targets*	Type	Description
RulesFiles	BabelRulesFiles	String	Semicolon-separated list of rules file paths
LogFile	BabelLogFile	String	Log file path
MapInFiles	BabelMapInFiles	String	List of XML map files to process
MapOutFile	BabelMapOutFile	String	File path of the output map file
KeyContainer	*KeyContainerName*	String	Key container name
KeyFile	*KeyOriginatorFile*	String	Path of the strong-name key file
KeyPwd	*SigningCertPassword*	String	Strong-name certificate password
TakeFiles	BabelTakeFiles	String	List of regular expressions to match against deployed XAP package assemblies that should be obfuscated - Tthis property is equivalent to the –take *Babel* command line switch.
SkipFiles	BabelSkipFiles	String	List of regular expressions to match against deployed XAP package assemblies that should **not** be obfuscated. This property is equivalent to the –skip *Babel* command line switch.
NoWarnings	NoWarnings	String	List of warning message codes to ignore
MergeAssemblies	MergeAssemblies	String	List of assembly files to merge into the primary assembly
Internalize	MergeInternalize	Boolean	Whether merged types have their visibility restricted to the assembly
CopyAttributes	MergeCopyAttributes	Boolean	Whether assembly-level attributes of merged assemblies are copied into the target assembly
EmbedAssemblies	EmbedAssemblies	String	List of assembly files to embed into the primary assembly
FlattenNamespaces	FlattenNamespaces	Boolean	Whether namespace names information should have their types' visibility restricted to the assembly
UnicodeNormalization	UnicodeNormalization	Boolean	Whether to obfuscate member names using Unicode symbols
ObfuscateTypes	ObfuscateTypes	Boolean	Whether to obfuscate the names of types
ObfuscateEvents	ObfuscateEvents	Boolean	Whether to obfuscate the names of events
ObfuscateMethods	ObfuscateMethods	Boolean	Whether to obfuscate the names of methods
ObfuscateProperties	ObfuscateProperties	Boolean	Whether to obfuscate the names of properties
ObfuscateFields	ObfuscateFields	Boolean	Whether to obfuscate the names of fields
VirtualFunctions	VirtualFunctions	Boolean	Whether to enable virtual member name obfuscation
OverloadedRenaming	OverloadedRenaming	Boolean	Whether to overload method names when possible
ControlFlowObfuscation	ControlFlowObfuscation	Boolean	Whether to enable MSIL control flow obfuscation
ILIterations	ILIterations	Int32	Number of iterations used in the control flow obfuscation algorithm.
EmitInvalidOpcodes	EmitInvalidOpcodes	Boolean	Whether to emit invalid op-codes
StringEncryption	StringEncryption	Boolean	Whether to enable encryption of user strings

Task Attribute	*Babel.Build.targets*	Type	Description
StringEncryptionAlgorithm	StringEncryptionAlgorithm	String	Name of the encryption string algorithm that will be used to encrypt strings
MsilEncryption	MsilEncryption	Boolean/String	Encrypt the IL of methods that match a given regular expression
ResourceEncryption	ResourceEncryption	Boolean	Whether to encrypt Resources
SuppressIldasm	SuppressIldasm	Boolean	Whether to prevent ILDASM from disassembling the obfuscated target
SuppressReflection	SuppressReflection	Boolean	Whether to prevent tools using reflection from displaying metadata.
DeadCodeElimination	DeadCodeElimination	Boolean	Whether to remove unused methods from the target assembly
XapCompressionLevel	XapCompressionLevel	Int32	XAP package compression level

Table 6 Babel task attributes

Advanced Topics

In this section we'll examine some advanced *Babel* usage topics like obfuscation of a Silverlight application package, signing an assembly with *Portable Information Exchange* key, and successfully obfuscating assemblies that target x64 *Windows* platforms. Following that is a description on how to store MSIL encrypted method data into an external file instead of a managed resource, and use that file for loading encrypted code at runtime. Another interesting technique is obfuscating the public members of an assembly that will be referenced in the main application; that is explained in the Obfuscate Multiple Assembly section. Finally, we'll give you some hints on how to troubleshoot problems that sometimes occur during the obfuscation process.

Silverlight XAP Packages

Silverlight applications are deployed by means of XAP packages. A XAP package is a zip compressed file that contains the application assemblies, resources and an application manifest *AppManifest.xaml* that defines the assemblies to be deployed. With *Babel* you can obfuscate the assemblies contained in the XAP package directly without working with individual assemblies outside the XAP package. *Babel* accepts as a primary assembly source the XAP package itself and performs obfuscation of all the deployed assemblies in a XAP package. The deployed assemblies are those listed as <*AssemblyPart*> element in the *AppManifest* XML file. You can also specify the assemblies that will be obfuscated and the ones that will be not, using the *–take* and *–skip* command line options. These two options accept a regular expression as an argument and those assembly names that will match the given expression will be taken or skipped by the obfuscator. Because the size of the XAP package affects the overall download time required to deliver the application to the client, you can tweak the XAP package size by changing the compression level using the *–compress* option.

Satellite Assemblies

Satellite assemblies are those generated by *Visual Studio* for projects that contain localized resources. These localized assemblies contain only resources—no code. *Babel* can handle and automatically obfuscate satellite DLLs that are part of an obfuscated executable. You don't need to specify them on the command line as inputs. *Babel* will search for assemblies' culture-specific resources located in application directories named after their localized culture names. For example, the application directory "*Math"* that contains *Calculator.exe,* should contains a directory *"it"* which contains the satellite assembly localized for the Italian culture named *Calculator.resources.dll*. *Babel* will rebuild the localized resource directories tree in the output folder.

Strong Name Assembly Signing Using PFX Key

Babel supports assembly re-sign with strong name key and PFX (*Portable Information Exchange*) file formats. Also, the key container can be used to re-sign an obfuscated assembly. The re-signing is performed at the end of the obfuscation process after the Emit phase. The command-line options that control the re-sign process are: *–keyfile, –keyname*, and *–keypwd*. The *keyfile* option is used to set the key-pair file path. *Babel* recognizes whether the key pair is a strong name key (*.snk*) or a Portable Information Exchange (*.pfx*) from the file extension. The *.snk* format does not require any other information, whereas *.pfx* files need to be unlocked by a password that should be specified using the *--keypwd* command-line option. If this option is not specified in the command line, *Babel* will prompt the user to enter the proper password during the Emit phase.

Obfuscating x64 Assemblies

Depending on the target platform, the *Visual Studio* compiler emits the desired CPU architecture into the PE header. Users of *Visual Studio* can set the target platform in the build project's property page. The default platform is "*Any CPU*" which indicates that the compiled assembly can run on any version of Windows. When a managed executable is loaded, Windows examines the PE header to determine whether the application requires 32-bit or 64-bit CPU architecture to ensure that it matches the computer's CPU type. Windows ensures the compatibility of 32-bit applications on 64-bit CPUs with a technology called WoW64 (Windows on Windows64) that emulates 32-bit instruction sets on a 64-bit CPU, albeit with performance loss. When the application architecture is known, Windows loads the framework execution engine Just-In-Time compiler (JIT) specific to the executable target platform (x86, x64, Itanium). On *x64* operating systems, the MSIL code of assemblies targeting "*Any CPU" platform* is validated by the .NET Framework runtime before execution.

To protect an assembly from being disassembled by tools like *.NET Reflector*, *Babel* can inserts into any MSIL method invalid byte codes that do not correspond to any MSIL op-code instruction (--invalidopcodes). The resulting assembly is no longer IL verifiable and Windows can execute it only on a WoW64 subsystem. Suppose that the obfuscated assembly is a DLL referenced by an executable that runs in full 64-bit environment. When the runtime tries to load the DLL obfuscated with invalid op-codes, it checks the CPU JIT requirements, and because they do not match, it throws an *InvalidProgramException* exception. The only way to generate obfuscated DLLs or executables fully compatible with x64 is to disable the injection of invalid op-codes during obfuscation.

Customize MSIL Encryption

With MSIL encryption the method code is hidden in a *.NET Framework DynamicMethod* object compiled at runtime with encrypted code stored in a managed resource. The original method is reduced to a call to the compiler engine stub. The *DynamicMethod* objects compiled with encrypted MSIL code are executed and then discarded. Moreover, once compiled, they cannot be modified. This makes the reverse engineering of encrypted methods very difficult. Commonly, the encrypted code is embedded into the assembly resources, but *Babel* can also store the encrypted MSIL code in standard files that can be loaded at runtime by the dynamic method compiler, that is embedded into the obfuscated assembly. This means that encrypted methods are stripped out from the assembly and serialized into an external file. The obfuscated assembly contains only calls to the compiler stub and there is no way to get the original method code without the encrypted file. Those files can be used to implement a custom licensing method or even stored in a database or secure store, and retrieved by the application when required.

To configure the generation of encrypted files, *Babel* needs to know the encrypted source name and the method that should be called to retrieve the encrypted data. The former comes from XML rules files; the latter from custom attributes. We have already seen in the MSIL Encryption section the XML properties available for the "*msil encryption*" feature: *Cache*, *MinInstructionCount*, *MaxInstructionCount* and *Source*. *Source* contains a string value that is the name of the encrypted data source associated with all the methods encrypted by the rule. Consider the following XML rule:

```
<Rule name="Rgb" feature="msil encryption" exclude="false">
  <Target>Methods</Target>
  <Pattern>Utilities.Rgb::*</Pattern>
  <Properties>
    <Cache>true</Cache>
    <Source>rgb</Source>
    <MinInstructionCount>5</MinInstructionCount>
  </Properties>
  <Description>Encrypt methods of Rgb class.</Description>
</Rule>
```

This rule tells *Babel* to encrypt all the *Rgb* class methods that contain at least five MSIL instructions and to assign the source name *rgb* to the encrypted data. *Babel* will generate in the output folder a file named *rgb.eil* (where .eil stand for Encrypted IL) that contains the encrypted code that will be used by the dynamic method compiler at runtime to generate the *Rdb* class methods.

Once you have the encrypted data files you need to tell Babel the method that will be used to retrieve the encrypted stream at runtime. This can be done by using the custom attribute *System.Reflection.ObfuscationAttribute* on an internal static method that takes the source name string as a parameter and returns the encrypted MSIL *System.IO.Stream* object. The *System.Reflection.ObfuscationAttribute* must specify the constructor-named parameter *Feature* as `"msil encryption get stream"`.

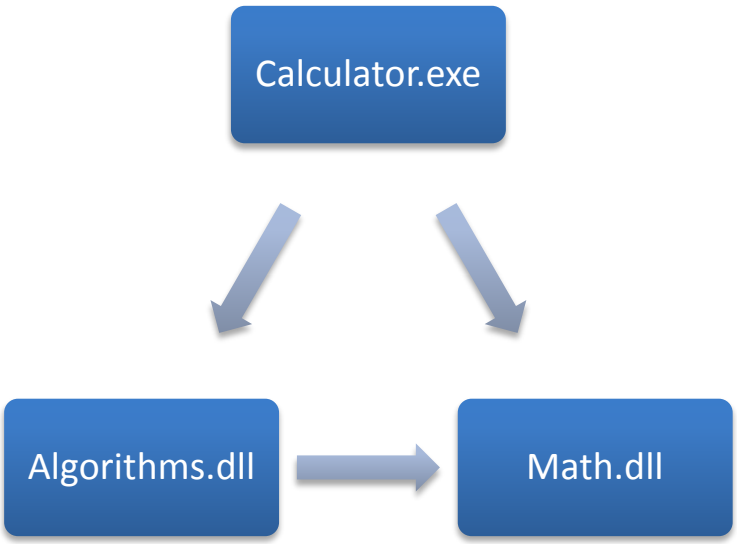
```
/// <summary>
/// Given the MSIL source name, returns the encrypted code stream.
/// </summary>
/// <param name="source">The name that identifies the source
/// stream.</param>
/// <returns>Encrypted MSIL source stream.</returns>
[Obfuscation(Feature="msil encryption get stream")]
internal static Stream GetSourceStream(string source)
{
}
```

At runtime, when an encrypted method of the *Rgb* class is called, the dynamic method compiler recognizes that the encrypted method data is associated with an external source instead of the assembly-embedded resource, and it will make a call to the method *GetSourceStream* with *source* argument equal to “rgb” to retrieve the encrypted *Stream* object.

Obfuscate Multiple Assemblies

The obfuscation process itself can only rename the internally visible assembly symbols, leaving the public members untouched. Otherwise the runtime execution of assemblies that consume the obfuscated types will be compromised. Babel can obfuscate public members in multiple assemblies, fixing the name of obfuscated symbols referenced in each assembly. Multiple-assembly obfuscation is made possible by means of map files. Map files are XML files produced as output from the obfuscation process, and contain all the associations between original and obfuscated symbol names. When using map files, it is possible to recover the original symbol names from the obfuscated ones. Map files must not be deployed with the obfuscated application under any circumstances. Keep them private.

Consider the following example of a managed application (*Calculator.exe*) that references two assembly libraries: *Algorithms.dll* and *Math.dll*. Suppose that *Algoritms.dll* references *Math.dll* also. The reference graph is represented as follows:



The two referenced assemblies: *Algorithms.dll* and *Math.dll are* general-purpose assemblies, used in several other projects. We want to obfuscate the entire project without merging the referenced libraries, as they are used by other applications as well and not just *Calculator.exe.* We want to keep the deployed package size as small as possible, and exclusion of referenced assemblies will assist with that. Algorithms and Math assemblies contain many public classes, so to effectively obfuscate the whole project we'd like to mangle public members as well. Babel can address this issue by generating an obfuscation map file for each referenced assembly that will have its public member obfuscated and use those map files when obfuscating the main application assembly or every other assembly with references back to public obfuscated assemblies.

To set up the obfuscation process, first we need to tell *Babel* to obfuscate all public symbols of *Math.dll* and *Algorithms.dll* assemblies. This can be done by inserting into the source code the following assembly-level custom attribute declaration:

```
[assembly: System.Reflection.ObfuscateAssembly(true)]
```

This attribute provides a way to configure *Babel* to obfuscate public symbols without using external configuration files. Setting the Boolean constructor parameter to true specifies that the assembly is private so that public member names can be safely obfuscated. If you don't want to change the source code you may even create an XML rules file that forces the obfuscation of public member names:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  publicRules.xml
-->
<Rules version="2.0">
    <Rule name="Obfuscate public" exclude="false">
        <Access>Public</Access>
        <Pattern isRegEx="false">*</Pattern>
        <Description>Obfuscate all public symbols.</Description>
    </Rule>
</Rules>
```

Once we configure obfuscation of public symbols we are ready to start *babel.exe*, including in the command line the option --mapout whenever we need to produce an output map file, and --mapin for each assembly that has a reference to a previously obfuscated assembly. We need to start from the assemblies that have no references to other obfuscated one. Returning to our *Calculator* example we can enter into a DOS shell the following commands:

```
babel.exe Math.dll --rules publicRules.xml –-mapout

babel.exe Algorithms.dll --rules publicRules.xml –-mapout
          --mapin Math.dll.map.xml

babel.exe Calculator.exe –-mapin Math.dll.map.xml --mapin Algorithms.dll.map.xml
```

Using this obfuscation feature it is possible to achieve a high level of renamed symbols, comparable to the level obtained by merging referenced assemblies.

Troubleshooting a Broken Application

Every effort has been made to avoid runtime problems caused by obfuscation. But occasionally obfuscation can break an application, or cause it to function in an unexpected manner. In this situation there are several things that can be done. If you run a full obfuscation that involved renaming, string encryption, control flow obfuscation, MSIL encryption and so on, try to lower the obfuscation level by running *Babel* with fewer tasks to perform.

For instance, consider disabling only string encryption, and then check whether the obfuscated application is fixed. If not, disable another phase like MSIL encryption, and then check again. If the problem persists, you can disable the remaining options one by one. For example, to disable type obfuscation, enter in the command line the option *–notypes* or its short form *–not*. If this fixes the application you will know that the problem is with type renaming. Of course you can go on and disable all the other renaming options: *-notmf* command will disable the renaming of types, methods and fields, leaving only events and properties obfuscation. When you find the cause of the problem, you can refine your search by running the *peverify.exe* tool on the obfuscated target. The hints that come from *peverify* should be enough to spot the cause of the error and give you the necessary information to make a custom XML rule to avoid the obfuscation of the faulty symbol.

There is another possibility of course: you found a bug in *Babel*. In this case, contact email support:

support@babelfor.net

Send all the useful information you can, such as:

- Operating system you are running on and processor architecture (*x86* or *x64*)
- *Babel* command line used
- *Babel* log output (use the *--logfile* command line option)
- *Peverify.exe* output
- Steps required to reproduce the issue

The more information you send, the more likely it will be that we can reproduce and fix the problem. If possible, you should also send the faulty assembly or, if this is not possible, a sample Visual Studio project that reproduces the same issue. This last item is not always possible, but it would help us tremendously in solving the problem.

Tips to Obfuscate Better

This paragraph will give you a list of useful tips to protect better your application using Babel Obfuscator.

1. ***Consider to declare as many types as possible internal (VB Friend)***: Public types and methods that are externally visible will not be renamed because they will be potentially used by other assemblies. Internal (or Friends for VB) declared types are accessible from within the assembly and are not visible to external assemblies so they can be safely obfuscated.

2. ***Enable always the Agent***: The Agent can help you to avoid common obfuscation problems that sometimes cause the obfuscated application to crash. It's also useful to look at the output produced by the Agent when you need to fine tune your XML rule files.

3. ***Use hash algorithm when encrypting strings***: This will encrypt your strings binding the encrypted data to strong name signature.

4. ***Enable dead code removal***: At least this will spotlight unused method and give you some hint to refactoring your code.

5. ***Do not MSIL encrypt too many methods***: MSIL encryption is a powerful protection feature, but it can slow down significantly the obfuscated application. Encrypt only those methods that run a limited number of times and contains code that you want to secure.

6. ***Encrypt resource when necessary***: resource encryption can hide all the embedded resource into your assembly but it comes at a cost of decryption during execution. This will increase the time needed by your application to access resources the first time.

7. ***Consider the environment your application is targeting***: If your application is targeting x64 platform, don't use invalid op-codes option because it will cause your application to run on WOW64.

8. ***Sign with strong name your assemblies***: Babel bind encrypted data to assembly strong name and insert custom code to detect data tampering.

9. ***Run peverify.exe /MD on the obfuscated assembly***: This tool can tell you if the metadata of your obfuscated application is verifiable.

10. ***Check the Babel command line output***: The Babel output can give you a lot of information about the obfuscation process and display warnings that should be always reviewed.

11. ***Test your obfuscated application***: Obfuscation might break your code. To avoid any unexpected behavior at runtime perform all the necessary tests on the obfuscated assemblies.

Appendix A

Warning Codes

The following table shows the list of warning codes and messages.
The codes can be used with command line option --*nowarn*.

Code	Message
CF0001	Unknown invalid opcode mode: '{0}'.
EA0001	Assembly embedding is not supported on Silverlight assemblies.
EI0001	Missing key information, the target assembly strong name was removed.
EM0001	Methods that return pointers are not supported. {0}
EM0002	Methods with pointer parameters are not supported. {0}
EM0003	Methods with ref or out parameters are not supported. {0}
EM0004	Generic instance method inline tokens are not supported. {0}
EM0005	Inline metadata tokens are not supported. {0}
EM0006	Bad inline token type.
EM0007	Unknown metadata signature calling convention.
EM0008	Generic open types are not supported. {0}
EM0009	Generic parameter types are not supported. {0}
EM0010	Type not supported. {0}
MG0001	Type {0} from assembly {1} already exists in the target assembly.
OB0001	Warning unresolved virtual {0}: {1}
OB0002	Missing key information, the resource assembly '{0}' strong name was removed.
PR0001	Target already processed by Babel; some problems may occur at runtime or during obfuscation.
PR0002	{0} warning: missing {1} string method.
PR0003	Suppressing MSIL disassembler option is not supported on .NET Framework {0} assemblies.
PR0004	Suppressing MSIL disassembler option is not supported on Silverlight 2 assemblies.
PR0005	Duplicate encrypt string method found {0}.
PR0006	Duplicate decrypt string method found {0}.
RE0001	Resource encryption is not supported on Silverlight assemblies.
RU0001	Rule '{0}' has an unknown property name '{1}'.
RU0002	Rules file unknown node {0} at line {1}, position {2}, node {3} '{4}'.
RU0003	Rules file unknown attribute at line {0}, position {1}, attribute {2}='{3}'.
RU0004	Unknown feature '{0}'.
SE0001	String algorithm '{0}' is not supported on .NET Framework {1} assemblies. Algorithm '{2}' will be used instead.
SE0002	String algorithm '{0}' is not supported on Silverlight assemblies. Algorithm '{1}' will be used instead.

Index

V

W

X